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 The Idea of AME Coding

2.1 Compression is Equivalent to Prediction

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

Both huffman coding and fano coding are based on the assumption that the source is *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 [2, 3], the last step of 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 [4]. 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. The markov chain is built with the encoding/decoding process, so it is called "adaptive".

2.2 Mechanism of AME Coding

We assume no prior knowledge of the source. So the predictor is built simultaneously with the encode/decode process. We used markov chain to model the process. With the inspiration in [1], we use a tree (a weighted digraph in formal terms) to represent the markov process. The algorithm for AME encoding and decoding are shown in Algorithm 1 and 2 in Appendix A.1 and A.2. The codes are written in Python and can be found in the appendix A.3 and A.4.

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

```
catcatme
```

The building process is shown in Figure 1. It is summarized as follows:

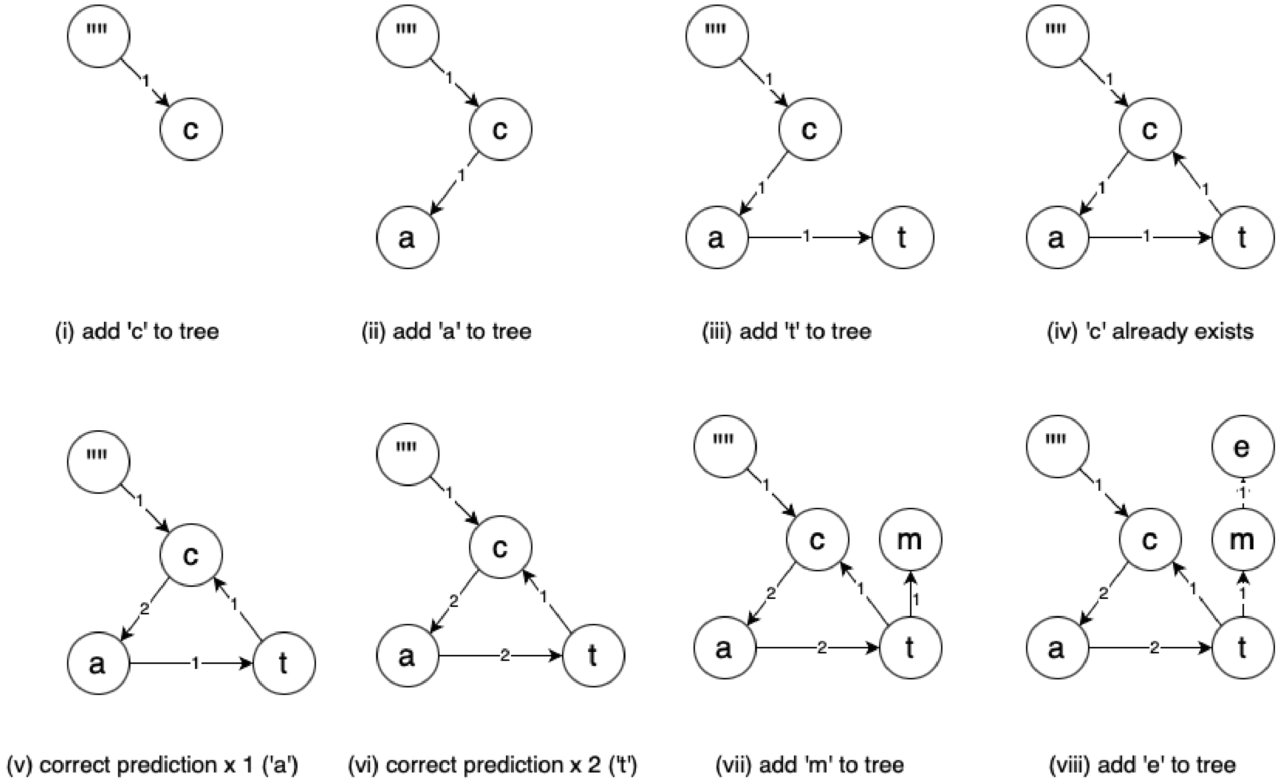
1. Initially, the tree is empty. The first character in the text is `'c'`, so we add `'c'` to the AME tree, mark the weight 1 (since first appearance) and add it directly to the output buffer.

2. 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.

3. Repeat this process until we could possibly "predict" the next character, i.e., the arrow with the highest weight in the tree from the current character leads to the correct character appeared in the text (If there are multiple arrows with the same highest weight, we take the most recent one).

4. The first correct prediction is the 5th character (`'a'`). We increase the weight from `'c'` to `'a'` by 1 and continue to predict the next character.

5. The next prediction (`'t'`) is also right. We increase the weight from `'a'` to `'t'` by 1 and continue to predict the next character.



(i) add 'c' to tree (ii) add 'a' to tree (iii) add 't' to tree (iv) 'c' already exists

(v) correct prediction x 1 ('a') (vi) correct prediction x 2 ('t') (vii) add 'm' to tree (viii) add 'e' to tree

Figure 1: The AME tree construction precedure

6. But the next two characters after `‘t’` are `‘m’` and `‘e’`, which are not even in the tree! Thus we first add the number of correct predictions in history (which is 2) and then add the two new characters (`‘m’` and `‘e’`) 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 E.3.

3 Entropy Coding Review

3.1 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 [5, 6]. 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 B.1.

3.2 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 C.1 and C.2.

4 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 [6, 7]. The second is to combine AME coding with Huffman and Fano coding and evaluate the performance of the new scheme.

4.1 Without AME

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

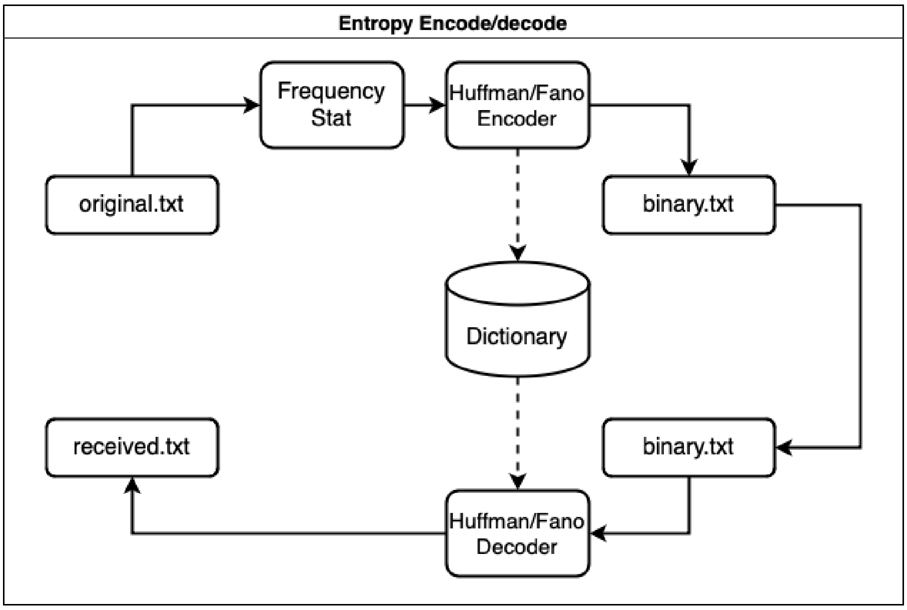


Figure 2: Flowchart of the experiment without AME

The content in `original.txt` contains the first three chapters of *the Game of Thrones*. Part of the `original.txt` is shown in Appendix E.1.

In order to perform huffman/fano coding on the text, we first need to calculate the frequency of each character in the text. According to statistics, the distribution of all characters in shown in Figure 3. The detailed frequency table of each character is shown in Appendix D.1.

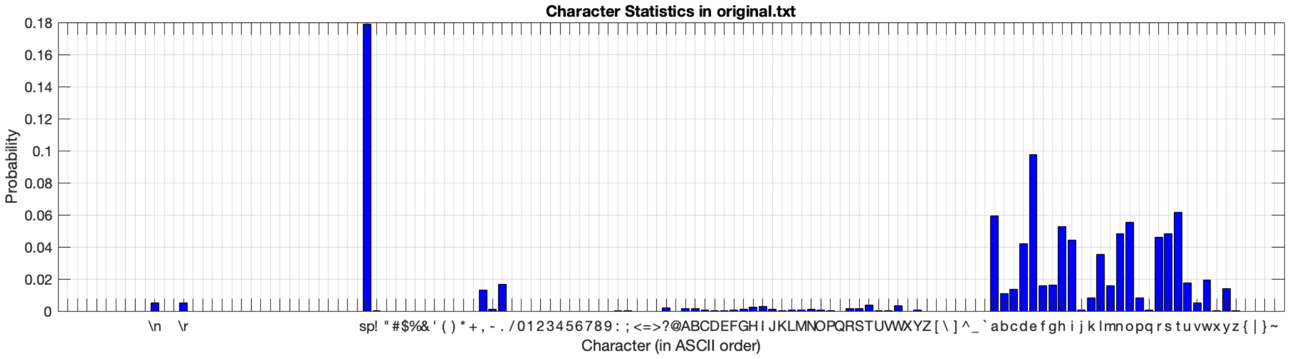


Figure 3: Distribution of each character in the `original.txt`

According to the frequency of each character, we can create the Huffman/fano tree based on the methods in Section 3.1 and 3.2 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 E.2. Then we send the encoded text (`binary.txt`) to the receiver and decode it using the same dictionary to recover the original text.

4.2 With AME

The experiment procedure of the encode/decode process in shown in Figure 4.

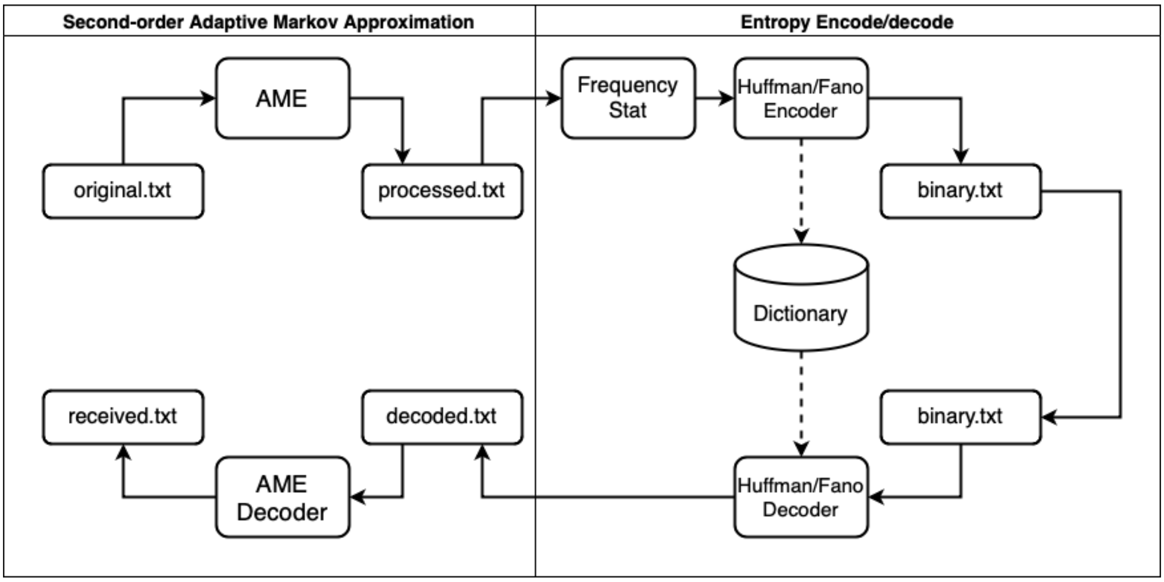


Figure 4: Flowchart of the experiment with AME

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

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 Section 2.2, we get the AME-encoded text (`processed.txt`). Part of the content of (`processed.txt`) is shown in Appendix E.3.

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. For example, there are no numbers in `original.txt` but there are in `processed.txt`. According to statistics, the distribution of all characters in shown in Figure 5. The detailed frequency table of each character is shown in Appendix D.2.

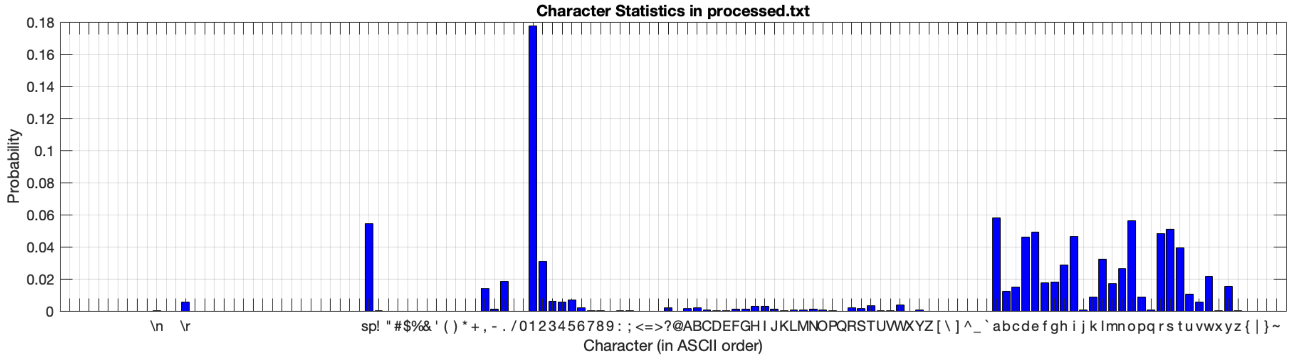


Figure 5: Distribution of each character in the `processed.txt`

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.

5 Results and Discussion

5.1 Performance comparison between Huffman and Fano coding

The four main metrics of Huffman and Fano coding are shown in Table 1.

Table 1: Performance Metrics for Huffman and Fano Coding

Parameter	Symbol	Huffman	Fano
Average Code Length	$\bar{L}$	4.5023	8.4244
Code Rate	R	0.9909	0.5282
Efficiency	η	0.9909	0.5282
Compression Rate	ξ	0.5628	1.0530

The four metrics are defined in Appendix F.1. Table 1 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* 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.

5.2 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 of 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 2 shows how using AME could save the length of the binary file.

Table 2: AME shortened the length of the binary file

Parameter	**Symbol**	**With AME**	**Without AME**	**Improved**
Huffman Binary Length	L_h	207797	217014	5.25%
Fano Binary Length	L_f	209343	218913	4.37%

This result shown that AME + Huffman/Fano coding could save about 5.25% and 4.37% of the length of the binary file, respectively.

6 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,” *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379423, July 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

A AME Coding Scheme

A.1 AME encode 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 → 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
```

A.2 AME decode algorithm

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
```

A.3 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:
        # Initialize the character in the tree with transition data
        tree[from_char] = {"transitions": {}, "highestFrequency": 0, "probableNextChar": ""}

    transitions = tree[from_char]["transitions"]
    if to_char in transitions:
        # Increment frequency for an existing transition
        transitions[to_char] += 1
    else:
        # Add a new transition to the tree
        transitions[to_char] = 1

    if transitions[to_char] >= tree[from_char]["highestFrequency"]:
        # Update the most probable next character if frequency increases
        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 empty if no prediction available
    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()  # Read and decode source file

    tree = {}  # Transition tree
    encoded_output = []  # Encoded text storage
    correct_predictions = 0  # Counter for consecutive correct predictions

    for i, current_char in enumerate(source_text):
        if i == 0:
            # Special case for the first character
            add_to_tree(tree, "", current_char)
            encoded_output.append(current_char)
        else:
            prev_char = source_text[i - 1]  # Previous character in sequence
            prediction = predict_next(tree, prev_char)

            if prediction == current_char:
                # Count correct predictions for consecutive matches
                correct_predictions += 1
            else:
                if correct_predictions > 0:
                    # Append count of correct predictions if any
                    encoded_output.append(str(correct_predictions))
                    correct_predictions = 0
                encoded_output.append(current_char)  # Append actual character
            add_to_tree(tree, prev_char, current_char)  # Update tree with transition

    if correct_predictions > 0:
        # Append remaining correct predictions at the end
        encoded_output.append(str(correct_predictions))

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

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

A.4 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:
        # Initialize the character in the tree with transition data
        tree[from_char] = {"transitions": {}, "highestFrequency": 0, "probableNextChar": ""}

    transitions = tree[from_char]["transitions"]
    if to_char in transitions:
        # Increment frequency for an existing transition
        transitions[to_char] += 1
    else:
        # Add a new transition to the tree
        transitions[to_char] = 1

    if transitions[to_char] >= tree[from_char]["highestFrequency"]:
        # Update the most probable next character if frequency increases
        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 empty if no prediction available
    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()  # Read and decode the encoded file

    tree = {}  # Transition tree
    decoded_output = []  # Storage for decoded output
    i = 0  # Index for traversing encoded text

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

        if i == 0:
            # First character, no previous context
            decoded_output.append(current_char)
            add_to_tree(tree, "", current_char)  # Add transition from initial state
        else:
            prev_char = decoded_output[-1]  # Previous character in the decoded output
            if current_char.isdigit():
                # Handle repetition encoded as numbers
                num_str = ""
                while i < len(encoded_text) and encoded_text[i].isdigit():
                    num_str += encoded_text[i]  # Build the number as a string
                    i += 1
                i -= 1  # Adjust index after loop
                repeat_count = int(num_str)
                for _ in range(repeat_count):
                    # Predict and decode repeated characters
                    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)  # Update tree with prediction
                    decoded_output.append(prediction)
                    prev_char = prediction
            else:
                # Handle regular character encoding
                decoded_output.append(current_char)
                add_to_tree(tree, prev_char, current_char)  # Update tree with transition

        i += 1  # Move to the next character in the encoded text

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

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

B Huffman Tree Visualization

B.1 Huffman Tree Generated for `original.txt`

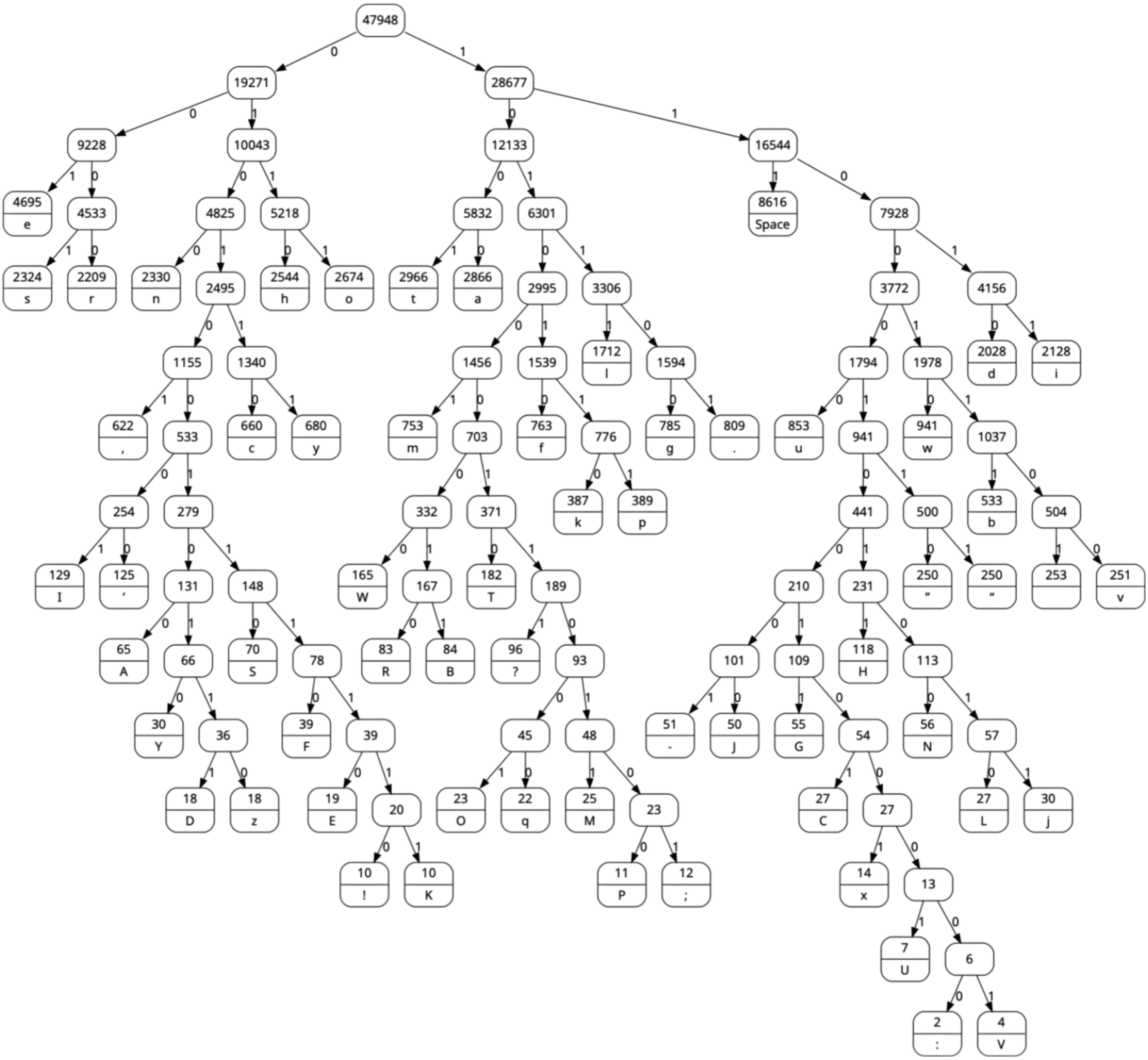


Figure 6: Huffman tree visualized for `original.txt`

B.2 Fano Tree Generated for `processed.txt`

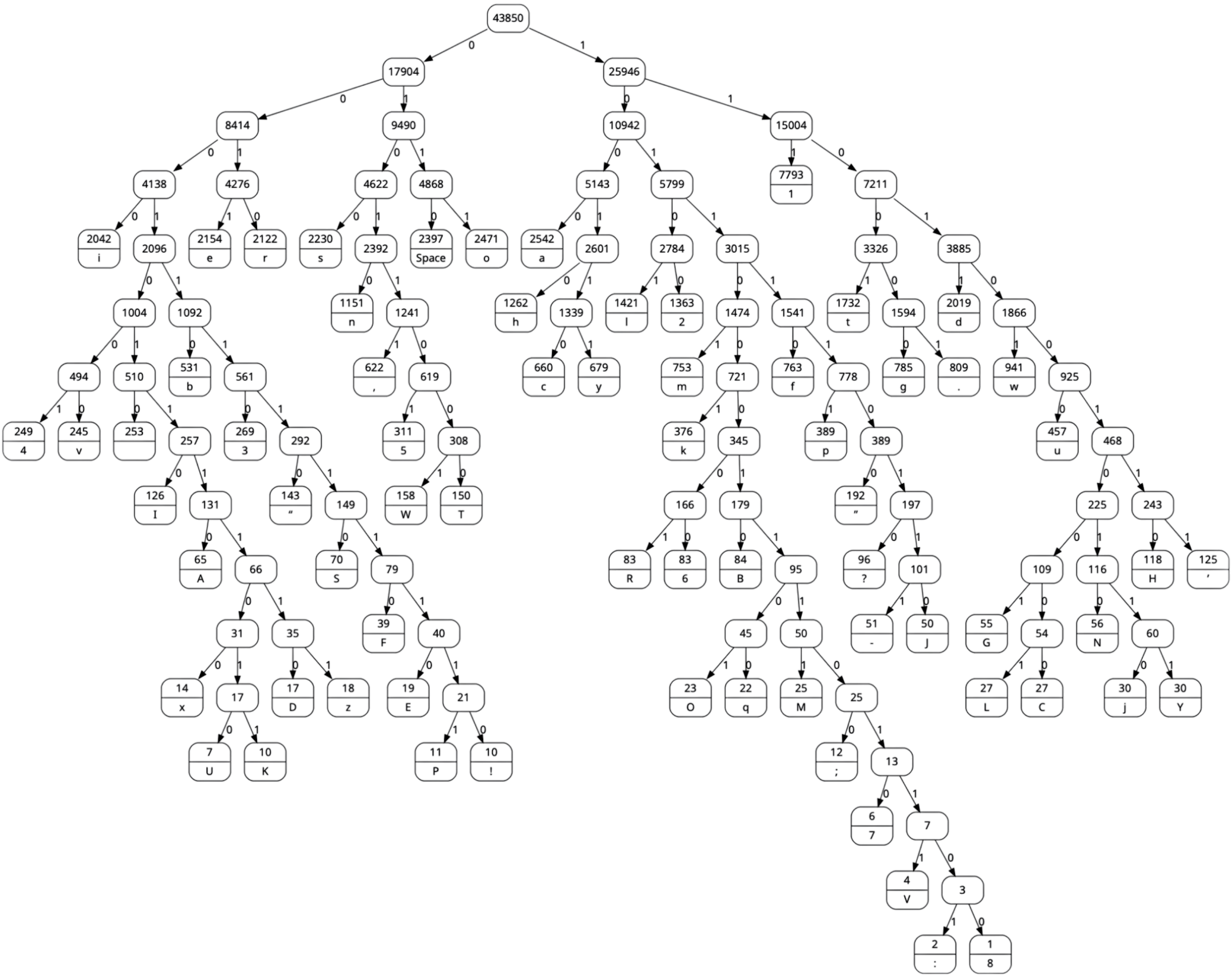


Figure 7: Huffman tree visualized for `processed.txt`

C Entropy Coding Scheme

C.1 MATLAB code for Huffman encode/decode procedure

Listing 3: Huffman encode procedure (without AME example)

```matlab
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)

```matlab
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

```matlab
% 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);
```

```matlab
fclose(fileID);

% 4.put the dictionary in a .txt file and display that
fileID = fopen(dictFilePath, 'w');
for i = 1:length(dict)
    fprintf(fileID, '%s: %s\n', num2str(cell2mat(dict(i, 1))), num2str(cell2mat(dict(i, 2))));
end
fclose(fileID);

% 5.decode the encoded text and put that in a .txt file using Huffman decoding
decodedMessage = huffman_decode(encoded_text, dict);
fileID = fopen(decodeFilePath, 'w');
fprintf(fileID, '%s', decodedMessage);
fclose(fileID);

% 6.verify if decoding was successful
if isequal(text, 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);
```

C.2 MATLAB code for Fano 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 = [buffer, encodedMessage(i)]; % Append bit to buffer

        if isKey(codeMap, buffer)
            % If the buffer matches a code in the dictionary
            decodedChar = codeMap(buffer);
            decodedMessage = [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, 'rb');
text = fread(fileID, '*char')'; % Read all characters, preserving spaces and
    line endings
fclose(fileID);
```

```

% 3. encode that original text using Fano decoding and put that in a .txt file
[encodedMessage, dict] = fano_encode(text);

fileID = fopen(encodedFilePath, 'wb');
fwrite(fileID, encodedMessage, 'char'); % Write the encoded message as is
fclose(fileID);

% 4. put the Fano dictionary in a .txt file with CRLF format
fileID = fopen(dictFilePath, 'wb');
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);

% write the decoded message to a file
fileID = fopen(decodedFilePath, 'wb');
fwrite(fileID, decodedMessage, 'char'); % Use fwrite to ensure exact output including spaces and line endings
fclose(fileID);

% 6. 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);
```

D Characters Statistics

D.1 Statistics in `original.txt`

Table 3: Frequency of each character in `original.txt`

ASCII	Char	Probability	ASCII	Char	Probability	ASCII	Char	Probability
10	\n	0.005249	32	sp	0.178751	65	A	0.001349
13	\r	0.005249	33	!	0.000207	66	B	0.001743
44	,	0.012904	45	-	0.001058	67	C	0.000560
46	.	0.016784	58	:	0.000041	68	D	0.000373
59	;	0.000249	63	?	0.001992	69	E	0.000394
70	F	0.000809	71	G	0.001141	72	H	0.002448
73	I	0.002676	74	J	0.001037	75	K	0.000207
76	L	0.000560	77	M	0.000519	78	N	0.001162
79	O	0.000477	80	P	0.000228	82	R	0.001722
83	S	0.001452	84	T	0.003776	85	U	0.000145
86	V	0.000083	87	W	0.003423	89	Y	0.000622
97	a	0.059459	98	b	0.011058	99	c	0.013693
100	d	0.042074	101	e	0.097405	102	f	0.015830
103	g	0.016286	104	h	0.052779	105	i	0.044148
106	j	0.000622	107	k	0.008029	108	l	0.035518
109	m	0.015622	110	n	0.048339	111	o	0.055476
112	p	0.008070	113	q	0.000456	114	r	0.045829
115	s	0.048215	116	t	0.061534	117	u	0.017697
118	v	0.005207	119	w	0.019522	120	x	0.000290
121	y	0.014108	122	z	0.000373			

D.2 Statistics in `processed.txt`

Table 4: Frequency of each character in `processed.txt`

ASCII	Char	Probability	ASCII	Char	Probability	ASCII	Char	Probability
10	\n	0.000023	32	sp	0.054662	65	A	0.001482
13	\r	0.005770	33	!	0.000228	66	B	0.001916
44	,	0.014184	45	-	0.001163	67	C	0.000616
46	.	0.018449	49	1	0.177715	68	D	0.000388
50	2	0.031083	51	3	0.006134	69	E	0.000433
52	4	0.005678	53	5	0.007092	70	F	0.000889
54	6	0.001893	55	7	0.000137	71	G	0.001254
56	8	0.000023	58	:	0.000046	72	H	0.002691
59	;	0.000274	63	?	0.002189	73	I	0.002873
74	J	0.001140	75	K	0.000228	76	L	0.000616
77	M	0.000570	78	N	0.001277	79	O	0.000525
80	P	0.000251	82	R	0.001893	83	S	0.001596
84	T	0.003421	85	U	0.000160	86	V	0.000091
87	W	0.003603	89	Y	0.000684	97	a	0.057969
98	b	0.012109	99	c	0.015051	100	d	0.046042
101	e	0.049121	102	f	0.017400	103	g	0.017902
104	h	0.028779	105	i	0.046567	106	j	0.000684
107	k	0.008574	108	l	0.032405	109	m	0.017172
110	n	0.026248	111	o	0.056350	112	p	0.008871
113	q	0.000502	114	r	0.048391	115	s	0.050854
116	t	0.039497	117	u	0.010422	118	v	0.005587
119	w	0.021459	120	x	0.000319	121	y	0.015484
122	z	0.000410						

E Sample File Content

E.1 Partial content in `original.txt`

`original.txt` contains the first three chapters of *the Game of Thrones*. Part of the content is shown below:

```
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? ...
```

E.2 Partial content in `binary_huffman.txt`

```
0101101010111001110010101101011001001000101010110101100100100100010010010111011
0011110011001010110010110010011101110000111010101000001111010000011000011100111
0011111110110000001010001111001010101100100110001011100000100100100011111111110
0011000000111111101010011000110000011111100000110101011000000111010001000000111
0111000000101001100101000111101100001101000000001010011111000001110000001100111
1111010110000001111111100000111110110011000001101010110010111010011000001011111
0101101110101100000011100010001000001100100000100101101010011100000111111111100
0000110110011100110010011001011100010101100101100001011111001111100110000000111
0101011000000110110011100110000010101111100100010100101001101101011000100100101
0000111101011010000101110000100111101110111100010011101011110010001011101111111
1000100111001100010010010010111000001111110010110011000110000001110001000110101
```

E.3 Partial content in `processed.txt`

The following content is the AME-encoded part of the content in Appendix E.1.

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

F Performance Analysis

F.1 Metrics Definitions for Entropy Coding

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):

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

F.2 MATLAB Code for Performance Analysis of AME

Listing 9: `Main` function to plot the curve

```matlab
clear;
% Performance analysis for the coding scheme when changing the length of the encoding content
inputFilePath = 'originalFull.txt'; % the original text

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

% Initialize array for graphing
text_size = 2:10000:length(text);
huffman_binary_length = zeros(1, length(text_size));
fano_binary_length = zeros(1, length(text_size));
ame_huffman_binary_length = zeros(1, length(text_size));
ame_fano_binary_length = zeros(1, length(text_size));

% Encoding on partial content

for i = 1:length(text_size)
    partialText = text(1:text_size(i));

    % Directly perform huffman coding
    [huffman_partial_encoded, ~] = huffman_encode(partialText);
    huffman_binary_length(i) = length(huffman_partial_encoded);

    % Directly perform shannon—fano coding
    [fano_partial_encoded, ~] = fano_encode(partialText);
    fano_binary_length(i) = length(fano_partial_encoded);

    % Use AME first then perform huffman coding
    ame_partial_encoded = ame_encode(partialText);
    [huffman_ame_partial_encoded, ~] = huffman_encode(ame_partial_encoded);
    ame_huffman_binary_length(i) = length(huffman_ame_partial_encoded);

    % Use AME first then perform shannon—fano coding
    [fano_ame_partial_encoded, ~] = fano_encode(ame_partial_encoded);
    ame_fano_binary_length(i) = length(fano_ame_partial_encoded);
end

% Plot the results
figure;
plot(text_size, huffman_binary_length, 'r', 'LineWidth', 2);
hold on;
plot(text_size, fano_binary_length, 'b', 'LineWidth', 2);
plot(text_size, ame_huffman_binary_length, 'g', 'LineWidth', 2);
plot(text_size, ame_fano_binary_length, 'm', 'LineWidth', 2);
```

```
xlabel('Length of Text (characters)');
ylabel('Length of Encoded Message (bits)');
title('Performance Analysis of Different Coding Schemes');
legend('Huffman Encoding', 'Shannon—Fano Encoding', 'AME + Huffman Encoding',
    'AME + Shannon—Fano Encoding');
grid on;
hold off;
```