Contents

1 Introduction **2**

2 Compression is Equivalent to Prediction **2**

3 Mechanism of AME Coding **3**

4 Entropy Coding Review **5**
4.1 Mechanism of Huffman coding 5
4.2 Mechanism of Fano coding 5

5 Implementation **5**
5.1 Without AME 5
5.2 With AME 6

6 Results and Discussion **7**

7 Conclusion **7**

A Python code for AME encoding **8**

B Python code for AME decoding **9**

C Part of the `original.txt` **10**

List of Figures

1 Flowchart of the experiment without AME 5
2 Flowchart of the experiment with AME 6

List of Tables

1 Frequency of each character in the text 5

Abstract

In this article, we 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

Huffman coding is one of the most well-known source coding methods, proposed by David A. Huffman in 1952. It is a coding method based on the greedy algorithm, capable of generating an optimal prefix code for a given set of symbols, ensuring that the length of each symbols code is inversely proportional to its probability of occurrence. The advantages of Huffman coding lie in its efficiency and optimization: it dynamically generates the optimal code based on the frequency of symbols and ensures uniquely decodable codes [2, 3].

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, like LZ77, (need more), 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. 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. The codes are written in Python and can be found in the appendix A and B.

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
```

4 Entropy Coding Review

4.1 Mechanism of Huffman coding

4.2 Mechanism of Fano coding

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. 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 1.

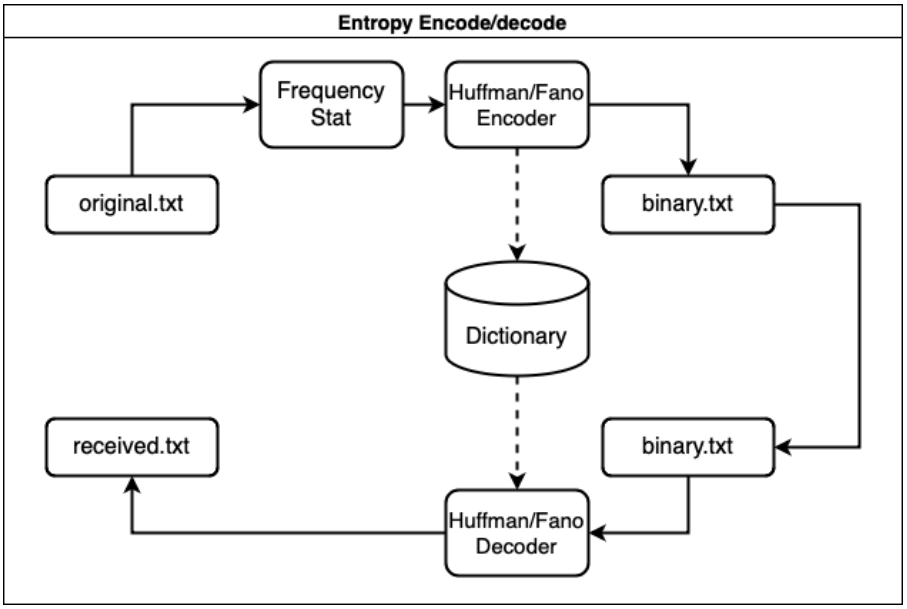

Figure 1: Flowchart of the experiment without AME

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

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

Index	Symbol	Probability	Index	Symbol	Probability	Index	Symbol	Probability
28	P	0.00022465	29	R	0.0017	30	S	0.0014
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

First, we applied the Huffman coding to the first three chapters of *the Game of Thrones* and calculate related 4 main parameters, including the average code length $\bar{L}$, code rate k, efficiency η, and compression ratio ξ.

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 significantly outperforms the Fano coding technique in terms of coding effectiveness [3, 4]. Therefore, we designed an exploratory section in which we applied the Fano coding technique to perform the same operations on the target text and compared the calculated parameters with those obtained using the Huffman coding technique.

5.2 With AME

The experiment procedure of the encode/decode process in shown in Figure 2.

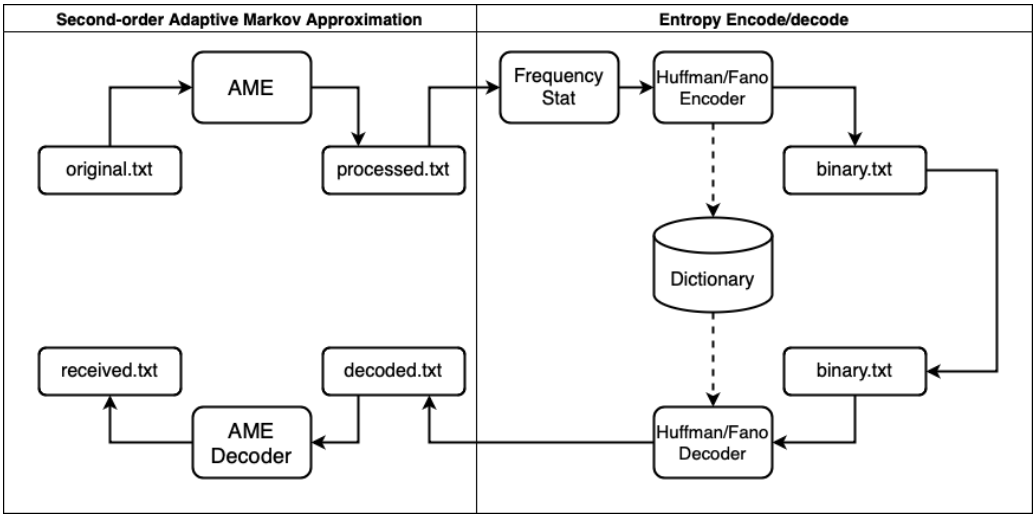

Figure 2: Flowchart of the experiment with AME

6 Results and Discussion

7 Conclusion

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.

A 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("source.txt", "markov_encoded.txt")
```

B Python code for AME decoding

Listing 2: AME decoder

```
# 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("markov_encoded.txt", "markov_decoded.txt")
```

C 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? ...
```