FULL TECHNICAL REPORT FOR 3F7 LABORATORY

DATA COMPRESSION: BUILD YOUR OWN CAMZIP

Yufeng Zhao

yz496@cam.ac.uk

Magdalene College

Summary:

In this report, we will explore methods to improve the performance of Arithmetic coding, which gives the best compression rate compared to Shannon-Fano coding and Huffman coding, as we found in the short lab. Specifically, we will investigate two variations of Arithmetic coding: adaptive arithmetic coding and context-based arithmetic coding. For each method, we will start with the principles and mechanisms behind it, followed by the implementation in MATLAB. Next, we will analyse the results and compare the performance with other codes. Finally, the practicality and limitations will be discussed.

1. Introduction

Arithmetic Coding has been found to have the best performance from the short lab in terms of compression rate compared with Shannon-Fano coding and Huffman coding, but it still has some potential defects and there are ways to improve. The first downside is that the encoder needs to scan the whole source and produce a probability model before encoding, and the decoder also requires the knowledge of the probability model to decode. This could be unachievable and undesirable in some occasion. Another problem is that the source to be compressed is not necessarily an independent and identically distribution. Exploiting this feature could improve the coding as well. These lead to Adaptive Arithmetic Coding and Context-based Arithmetic Coding which we will discuss in more details.

2. Adaptive Arithmetic Coding

2.1 Adaptive Arithmetic Coding

To introduce the idea of adaptive coding, we first look at a downside of the simpler arithmetic coding that we realised in the short lab. The traditional coding scheme requires the probability model to be precalculated and transmitted with source data to encode and decode successfully, which however, is undesirable or even not achievable in practice. This issue becomes significant especially for streaming data, i.e. data generated continuously by a large of data sources. Unlike images of texts, streaming data is usually transmitted as being generated, which makes it impossible for a first pass over the data to calculate a probability model.

To cope with this, the concept of *adaptive coding* is introduced. Adaptive coding is a coding method in which the probability model is defined only based on the existing source data. The encoder and decoder continuously update the model as it encodes/decodes, only according to the data it has already processed. Therefore, the transmission of a precalculated data model is no more necessary. The cost of adaptive model is that more computational power is required to update the probability model after each symbol encoded and the decoder will have to be more complex to keep in sync with the encoder. It is a safe prediction that when the source is long enough, the performance of adaptive coding will be close to the corresponding static version because the adaptive data model will converge to the real one relatively earlier.

The adaptive version of arithmetic coding applies the traditional interval coding methodology but without a predefined data model, instead probability is measured up to but not including the current symbol being encoded/decoded. This ensures that encoder and decoder will have the same frequency count for each symbol.

2.2 MATLAB Implementation

In actual implementation of adaptive arithmetic coding, rather than simply basing the probability model on the frequency counts of previous symbols, a starting bias is added. This is to avoid the problem that when a symbol appears for the first time, its probability will be zero which makes it not encodable. The key part of MATLAB program is presented below:

```
% initialise probability count
e = 1;
count = e * ones(256);
p = count / sum(count);
```

Here an initial bias of one is used, leading to a probability estimator called Laplace estimator. The effect of different value of starting bias `e` will be discussed later.

```
% update histogram
count(ind)=count(ind)+1;
p=count/sum(count);
```

After encoding or decoding a symbol, the frequency count is updated up to this symbol and the new probability model is calculated and normalised.

2.3 Results

	Entropy limit H(p)	Huffman Coding	Static Arithmetic Coding	Adaptive Arithmetic e=1	Adaptive Arithmetic e=0.01	Adaptive Arithmetic e=100
hamlet.txt	4.4499	4.47266	4.44991	4.50837	4.45315	6.94281
bible.txt	4.3428	4.38495	4.34275	4.34626	4.34315	4.63824
abrupt.txt	4.7082	N/A	4.70869	6.43966	4.78401	7.99539

*abrupt.txt is a dedicated file to test the performance of adaptive arithmetic coding on source strings with abrupt changes. It contains 5200 characters that start with 100 A's, followed by 100 B's, 100 C's, … , 100 Z's, and again 100A's, … 100Z's.

2.4 Observation and Discussion

For texts not changing drastically, adaptive arithmetic coding approaches the performance of static adaptive arithmetic coding with properly selected initial bias. It gives very good compression rate for long source file which matches our prediction. However, for source strings with significant change in patterns, adaptive coding does not perform as well as before because it is harder for the adapted estimation of probability to converge to the real model. A much longer run time than static arithmetic coding is also observed, which is natural due to the more complicated model and the higher algorithm complexity.

3. Context-based Arithmetic Coding

3.1 Context-based Coding

In all the previous experiments we conducted and compression ratios achieved, we assumed the text follows an independent and identically distribution (iid), which however, is obviously not the case for text in real life. The text of a language always has many certain patterns, for instance in English, we can almost expect a ‘u’ after a ‘q’ every time, that is, the conditional probability of a letter ’q’ being followed by a letter ‘u’ is almost 1. Exploiting this feature we can make our code much more efficient. This allows us to improve the performance of iid-model coding a lot by measuring the conditional probability distribution of the next symbol based on the context. This method is termed *context-based coding* or *contextual coding*.

3.2 MATLAB Implementation

For both the encoder and decoder, the core of applying this method is to establish a conditional probability table for each key existing in the source string. The approach is:

Scan the whole source string and find all unique keys (keySet) → Scan the source again to update the conditional probability distribution (p) → Normalise the probability model.

The following shows the MATLAB code for these steps:

```
% define keySet (previous context) and conditional probability

keySet=zeros(file_length-key_len,key_len);

for k = key_len:length(in)
    key = in((1+k-key_len):k);
    keySet(k-key_len+1, 1:key_len) = key;
end

keySet=unique(keySet,'rows');

% define conditional probability distribution

p = zeros(length(keySet),256);

for k = key_len+1:length(in)
    key = in((k-key_len):k-1);
    [dummy, ind] = ismember(key, keySet, 'rows');
    p(ind, in(k)+1)=p(ind, in(k)+1) + 1;
end

p=p/sum(sum(p));
```

The parameter here is the length of key K considered (key_len). One problem is how to store the keySet, which could be a very large array of vectors. Two methods were considered during the production of the program. The first one is simply representing all 256K possible keys, no matter if they exist in the source text. This method takes less space when K is small (K <= 2). The other way is to scan the whole source file and search for keys from the source, which will give (source length – key length) number of keys.

For example, if we try to encode ‘hamlet.txt’ which has 207039 characters, with 2 previous symbols considered (i.e. K = 2). With method 1, there would be 2562 = 65536 keys to be stored, whereas method 2 ends up giving 207039 – 2 = 207037 keys. However, if we consider more previous symbols, for example (K = 3), the number of keys for method 1 will significantly go up to 2563 = 16777216, which is obviously not practical. Yet in method 2, the number of keys is still close to the length of the source string. Furthermore, with the application of keySet=unique(keySet,'rows'); this number significant reduces to 1147 for K = 2, which is very acceptable. Therefore, method 2 is adopted in the final version of the program.

Another problem is that how to deal with the first few symbols which is even shorter than K. Here we decide to skip them and only take keys at length K. Correspondingly when decoding the first few symbols, only unconditional probability model is used since we do not have any keys at this length before the decoded symbol with relative section of code from decoder shown below:

```
for k = 1:ny     %for every symbol to be decoded

    %define conditional/unconditional p based on k
    if k<=key_len
        p = sum(prob)/sum(sum(prob));
        %use unconditional probability for the first few symbols
    else
        key = y(k-key_len:k-1);
        if size(key)~=[1,key_len]
            error('size');
        end
        [dummy,ind_key] = ismember(key,keySet,'rows');
        %use conditional probability defined by key
        p = prob(ind_key,:);
        p=p/sum(p);
    end
    .
    .
    .
```

For the decoder, it decodes a symbol by looking first at the previously decode symbols, assuming it has decoded the previous symbols correctly. Then it finds the key and uses the conditional probability as the probability model.

3.3 Result

Compression Rate / File	K = 1	K = 2	K = 3	K = 4	H(p)
Hamlet.txt	3.35303	2.35751	1.80453	1.38613	4.4499
Bible.txt	3.2691	N/A	N/A	N/A	4.3428

3.4 Observation and Discussion

We can find that the compression rate of context-based coding is lower than the entropy limit, which is an obvious result of exploiting the fact that text is not an iid model. The biggest drawback is that contextual coding requires enormous amount of processing time and large storage space. It takes impractically long time to complete encoding on MATLAB for hamlet.txt K >= 5, and bible.txt K >= 2. Applying contextual coding in other coding languages in some smarter ways may improve the speed but still it is undeniable that it is much slower than the iid arithmetic coding.

4. Conclusion

It is worth putting the two types of coding method considered in contrast together with static arithmetic coding:

Adaptive Arithmetic Coding is excellent when a precalculated probability model is unavailable or undesirable, at a cost of slightly inferior performance to static arithmetic coding, in terms of processing time and compression rate. However, it is still the best choice for streaming data. Context-based Arithmetic Coding is the only code that gives compression rates lower than the entropy limit. Though the shortcomings could be the even slower speed. Another limitation is that it is only advantageous to use contextual coding when the source follows some clear patterns.

It would be beneficial to combine Adaptive coding and Contextual coding to obtain the advantages of both. It is possible to improve the compression rate with contextual coding and speed the process and reduce the special complexity by adaptively updating the conditional probability model. In practice, context-based adaptive binary arithmetic coding(CABAC), has become the most powerful tool in video compression.

5. Reference

1. Amazon Web Service, What is Streaming Data? https://aws.amazon.com/streaming-data/

2. D Marpe, H Schwarz, T Wiegand, Context-based adaptive binary arithmetic coding in the H. 264/AVC video compression standard, IEEE Transactions on Circuits and Systems for Video Technology (Volume: 13, Issue: 7, July 2003)

3. Wikipedia, Lossless Compression, https://en.wikipedia.org/wiki/Adaptive_coding