Report of the Arithmetic Coding Homework

Chung-Yi Chen
311551070

Abstract

In this homework, I implemented arithmetic coding with different probability models including the fixed probability model and several variants of the prediction by partial matching (PPM) algorithms. The experimental results showed that arithmetic coding with the 32-bit symbol length plus second-order ppmb with the exclusion principle can achieve the best compression ratio of 3.6286, slightly better than 3.5925 achieved by the extended Huffman coding with the extended size of 2.

1 Implementation

For the concerns of execution speed and memory usage, the programming language used in this homework is C++. OpenMP [Ope21] is also used in this homework to accelerate the experiments by executing arithmetic coding with different probability models at the same time.

1.1 Arithmetic Coding

The implementation of arithmetic coding followed the integer implementation pseudocode described in [Say17]. The word length m of the lower bound l and upper bound u can be manually specified. This allows us to debug the program with human-calculated results by enabling the `show_step` flag to show the calculation steps of arithmetic coding. An example is shown in subsection 2.1.

The design of arithmetic coding and probability models are decoupled. The `encode` method of arithmetic coding only requires the probability model to implement the `get_prob` method and the `update` method to work. In other words, if a C++ class implements these two methods, its instance can be regarded as a valid probability model.

Take the fixed probability model as an example. In my implementation, the initialization (i.e., count the occurrence of each symbol) of the fixed model is outside the scope of the `encode` method of arithmetic coding. While encoding, the algorithm calls the `get_prob` method to get the cumulative probabilities for both the lower bound and upper bound. The `update` method is invoked at the end of each encoding iteration, and for the fixed model, it is just simply a return statement.

It is not hard to see that other probability models can be integrated into the implemented arithmetic coding with this design.

1.1.1 Implementation of Scaling

There are at least two possible implementations for the E_1/E_2 and E_3 scaling. The pseudocode is the first possible implementation, denoting it as impl-1 for convenience. Whether E_1/E_2 scaling happens or not, impl-1 always checks the E_3 condition and executes the E_3 scaling if true (i.e., two independent if-statements for checking E_1/E_2 and E_3 conditions). The second is the procedure in the provided solution of Exercise 6. It checks the E_3 condition only if the E_1/E_2 scaling did not happen (i.e., one if-else-if-statement for checking both conditions). Denote this implementation as impl-2. My implementation is impl-2, but can be changed to impl-1 with minor revisions to nine lines of code.

Here I want to show that two implementations can produce the same encoding. While checking l and u, there are always three possible situations to happen next.

Case 1. Neither E_1/E_2 nor E_3 condition holds.

Case 2. Either E_1/E_2 or E_3 condition holds.

Case 3. Both E_1/E_2 and E_3 conditions hold.

Both implementations have the same behavior for Case 1 and Case 2. Only Case 3 needs to be discussed. WLOG, say both E_1 and E_3 conditions hold, i.e., $0.25 \le l^{(k)} < u^{(k)} < 0.5$.

impl-1 If doing E_3 first, since the E_3 scaling is $E_3(x) = 2(x - 0.25)$, we can obtain a new interval $[0, 0.5)$ for $l^{(k+1)}$ and $u^{(k+1)}$. In the next iteration of impl-1, this falls into the E_1 condition and an E_1 scaling $E_1(x) = 2x$ is performed, and the next check for $l^{(k+2)}$ and $u^{(k+2)}$ could be any of the three cases. The transmitted sequence for E_3 and then E_1 is '01' and the composited scaling function $E_1\big(E_3(x)\big)$ is $E_1\big(E_3(x)\big) = 2\big(2(x - 0.25)\big) = 4x - 1$.

impl-2 If doing E_1 first, the new interval would be $[0.5, 1)$ for $l^{(k+1)}$ and $u^{(k+1)}$. In the next iteration of impl-2, the E_2 scaling $E_2(x) = 2(x - 0.5)$ must be applied, and the next check for $l^{(k+2)}$ and $u^{(k+2)}$ could be any of the three cases as impl-1. The transmitted sequence for E_1 and then E_2 is '01'. The composited scaling function $E_2\big(E_1(x)\big) = 2(2x - 0.5) = 4x - 1$.

Since the composited scaling function, $l^{(k)}$, and $u^{(k)}$ are the same for both implementations, $l^{(k+2)}$ and $u^{(k+2)}$ are the same. Also, the transmitted sequences of both implementations are the same. It is easy to see that this conclusion is also true when E_2 and E_3 conditions hold. This indicates that impl-1 and impl-2 have the same encoding for Case 3. Considering that impl-1 and impl-2 have the same behavior for Case 1 and Case 2, this means impl-1 and impl-2 can produce the same encoding for the same input sequence. In other words, impl-1 and impl-2 are equivalent. ■

1.2 Prediction by Partial Matching

The prediction by partial matching (PPM) algorithm has several variants. They differ on how to count the escape symbol for the first appearance of a symbol.

The original version is called ppma, no matter what symbol appeared how many times, the count of the escape symbol is always one. However, unlike ppma, when a symbol appears for the first time, rather than increasing the count of that appeared symbol by one, ppmb increases the escape symbol by one and keeps the count of that symbol zero. In ppmb, only if a symbol appeared twice before, its cumulative probability is used. Since this behavior is not aligned with ppma and ppmc, it introduces a bit of complexity in my implementation. ppmc can be considered as the combination of ppma and ppmb. When a symbol appears for the first time, ppmc increases not only that symbol but also the escape symbol by one.

Besides, the exclusion principle is also implemented for these three kinds of PPM. That is, there are six possible configurations (ppma, ppmb, and ppmc with exclusion or not) of PPM available for evaluation.

As for the interaction with arithmetic coding, just like the fixed probability model, the configuration of PPM is also done outside the scope of the `encode` method. Any modification to the PPM algorithm does not affect arithmetic coding. The `get_prob` method returns a list of pairs of cumulative probabilities. If the number of pairs n_p is greater than one, indicating that the escape symbol needs to be encoded $n_p - 1$ times. The contents of PPM are updated by a list of symbols passed via the `update` method invoked in the `encode` method. This again shows that the design of arithmetic coding is friendly for new probability models.

2 Evaluation

Let s and o denote the symbol length of arithmetic coding and the max order of PPM, respectively. For simplicity, (s, o) is used to describe an experiment of s-bit-symbol arithmetic coding with fixed probability model (fixed), o-order ppma (ppma), o-order ppma with the exclusion principle (ppma-e), o-order ppmb (ppmb), o-order ppmb with the exclusion principle (ppmb-e), o-order ppmc (ppmc), and o-order ppmc with the exclusion principle (ppmc-e). Additionaly, let $(s, o/p)$ short for $(s, o), (s, o+1), (s, o+2), \ldots, (s, p-1), (s, p)$.

As required, $(1, 0/2)$ and $(8, 0/2)$ are evaluated. Besides, thank to the efficiency of C++, $(1, 3/4)$, $(2, 0/4)$, $(4, 0/4)$, $(8, 3/4)$, $(16, 0/3)$, and $(32, 0/2)$ are also evaluated. Although my implementation allows arbitrary values of the max order of PPM, memory capacity is the major limitation. Executing these experiments took about 335 minutes on my Intel® Core™ i7-1260P with OpenMP enabled. There are 168 unique configurations of arithmetic coding in total.

2.1 Correctness

To check the correctness of my implementation, I took Problem 2 of Exercise 6 as a test scenario. Set the word length to 6 and use first-order ppma as the probability model to encode the sequence $cat\Delta ate\Delta hat$, compared with the provided solution, arithmetic coding can have (1) the same values of lower and upper bound until the first e, (2) the same values of cumulative probabilities at least until the first e, and (3) the same transmitted sequence until the first t. The calculation steps are available in `out.txt` and can be reproduced by compiling and running the program.

For (1) and (2), the inconsistency happens at the last e3 scaling while encoding e. The correct result should be $l = 001000$, $u = 110011$ but in the solution, it is $l = 000000$, $u = 110011$. For (3), the inconsistency happens due to a typo[1] in the solution, because the transmitted sequence is 1100101101 after encoding the first t, but later the sequence becomes 110000111 while encoding the first Δ.

The result of my implemented arithmetic coding is also double-checked with my manually calculated

[1]I ignored the fact that in the given solution, the E_3 bits are sent after all possible E_1/E_2 scaling is done while encoding a sybmol. The pseudocode sends the E_3 bits right after the first E_1/E_2. Even if there is no typo, the transmitted sequence will different when the E_3 counter is set and a series of E_1/E_2 are applied.

result. Although I only manually encoded the sequence *cat*Δ*ate*, it is enough to show that my implementation is correct.

2.2 Fixed Model vs PPMs

To further examine the characteristics of the fixed probability model and PPMs, two sequences are used. The first is a uniformly randomly generated sequence of 48 symbols, denoting rand-seq. It contains eight ‘0’, twelve ‘1’, fourteen ‘2’, and fourteen ‘3’. The second is a sequence of twelve repeated ‘0123’, denoting rep-seq. The actual sequences and their encoding can be found in `out.txt`.

Model	rand-seq	rep-seq
fixed	94 bits	96 bits
ppma	156 bits	30 bits
ppma-e	144 bits	30 bits
ppmb	138 bits	35 bits
ppmb-e	121 bits	35 bits
ppmc	158 bits	29 bits
ppmc-e	144 bits	29 bits

Table 1: Encoding lengths of the different probability model. The order of PPM is 2 and the word length of arithmetic coding is 6.

As Table 1 shows, fixed has almost the same lengths for rand-seq and rep-seq because these two sequences have similar PDFs, and PPMs are good at sequences with repeated patterns but bad at randomly generated sequences because the repeated patterns can be captured by PPMs (context-based compression techniques), while random sequences not.

By the way, let $\mathbf{x}$ be a sequence, and the theoretical lower bound of the encoding length $l(\mathbf{x})$ is given by $l(\mathbf{x}) = \lceil \log \frac{1}{P(\mathbf{x})} \rceil + 1$ bits. Also, let $\mathbf{x}_{\text{rand}}$ and $\mathbf{x}_{\text{rep}}$ represent rand-seq and rep-seq, respectively. We can find that both encoding lengths of fixed in Table 1 are smaller than their theoretical lower bounds $l(\mathbf{x}_{\text{rand}}) = 96$ and $l(\mathbf{x}_{\text{rep}}) = 97$. The reason is that the EOS symbol is not considered in my implementation, bringing the gap between the theoretical and actual encoding length.

2.3 Symbol Length vs Order

Since there are lots of data points and it is hard to place them into a single figure, I plotted them in six subfigures as Figure 1 shows.

One important observation is that a ppm algorithm with the exclusion principle can always gain a shorter encoding than the original version. For $(1, 0/4)$, the compression ratios of fixed and PPMs are all close to 1, meaning that the compression has no effect. For $(2, 0/4)$, the best compression

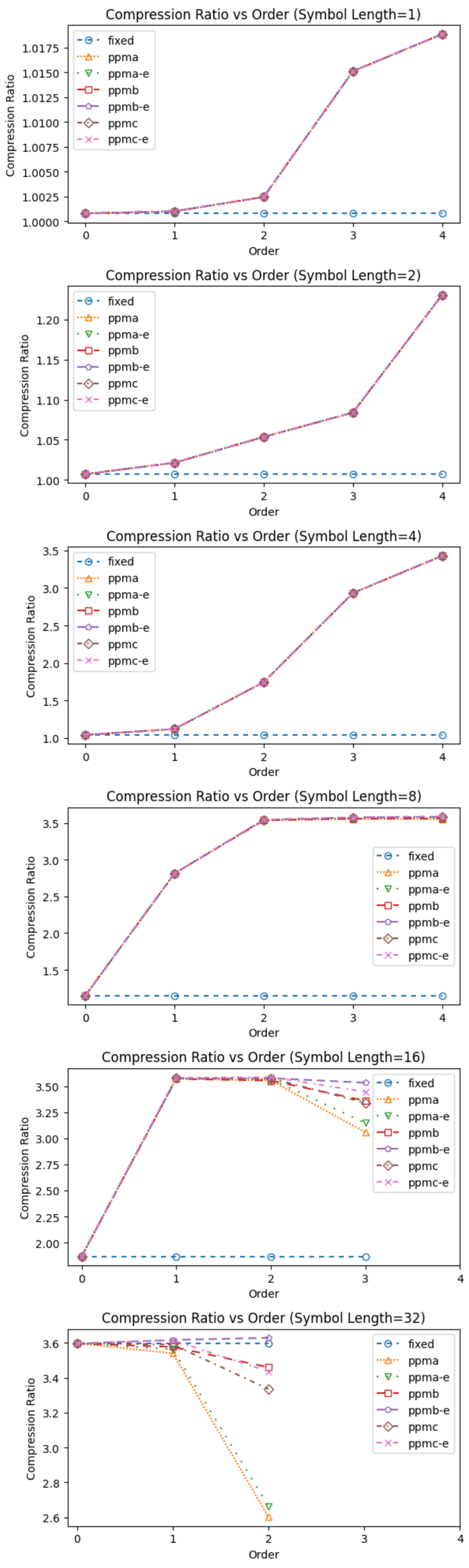

Figure 1: Compression ratio of different probability models.

ratio is 1.2307, the compression is still not that effective. However, when $s = 4$, the best compression ratio goes to 3.4322, which approaches the best compression ratio of 3.6286 achieved with the second-order ppmb-e plus a 32-bit symbol length. Based on Table 2, using a 16-bit symbol length can not achieve a better compression ratio of an 8-bit symbol length. If we do not consider the computation resource, memory resource, and time consumption, using second-order ppmb-e with a 32-bit symbol length is the best configuration. Nevertheless, if putting these factors into consideration, using the fixed model with a 32-bit symbol length is a more reasonable configuration, for it can achieve a compression ratio of 3.5981 and only took 7.07744 seconds to finish, as a comparison, the best configuration took 2730.05 seconds to finish. The dramatic difference may come from the cache misses.

	Best Config.		
Sym. Len.	Model	Order	Com. Ratio
1	ppma-e	4	1.0188
2	ppma-e	4	1.2307
4	ppma-e	4	3.4322
8	ppmc-e	4	3.5926
16	ppmc-e	2	3.5888
32	ppmb-e	2	3.6286

Table 2: Compression ratio of the best configuration given different symbol lengths.

For $s = 1, 2, 4, 8$, when o increases from 0 to 4, the compression ratios of all PPMs increase, and in the meantime, their values are very close. But things get different when $s = 16$. Almost all compression ratios of PPMs drop when $o = 2$. When $o = 3$, all compression ratios drop compared to $o = 1$. A much more surprising situation happens when $s = 32$. Only ppmb-e and ppmc-e can beat the fixed model when $o = 1$. when $o = 2$, only ppmb-e can beat the fixed mode. Almost all PPMs have performance degradation when o increases.

2.4 Compared with Huffman Coding

To be fair, the comparison of Huffman coding and arithmetic coding is separated into two parts. The first is to compare the compression ratio of static Huffman coding and arithmetic coding with the fixed probability model. The second is to compare the compression ratio of extended Huffman coding and arithmetic coding with the best configuration. The reason to use extended Huffman coding only is that adaptive Huffman coding cannot achieve a better compression ratio than extended Huffman coding, and static Huffman coding can be regarded as extended Huffman coding with an extended size of 1.

Sym. Len.	HC (static)	AC (fixed)
1	1	1.0008
2	1	1.0071
4	1.0302	1.0401
8	1.1448	1.1492
16	1.8624	1.8679
32	3.5889	3.5891

Table 3: Compression ratio of Huffman coding and arithmetic coding given different symbol lengths.

Sym. Len.	HC (ext.)	AC (best)
1	1/4	1.0188
2	1.0032/4	1.2307
4	1.0382/4	3.4322
8	1.1476/3	3.5926
16	1.8648/2	3.5888
32	3.5925/2	3.6286

Table 4: Compression ratio of extended Huffman coding and arithmetic coding with the best configuration given different symbol lengths. The value after the slash is the extended size n_e, which means the alphabets are extended $n_e - 1$ times.

As Table 3 shows, arithmetic coding always has a slightly better compression ratio than Huffman coding on all symbol lengths. The execution time of both algorithms is almost the same. Further, as Table 4 shows, with PPMs, arithmetic coding can outperform extended Huffman coding with the cost of having a much longer execution time.

3 Conclusion

The concise implementation of the arithmetic coding and probability models makes the pave for the efficiency of conducting considerable experiments in a reasonable time. The correctness has been checked, and the details are also discussed.

The experimental results show that context-based arithmetic coding is an effective compression algorithm, it can achieve a descent compression ratio with a smaller symbol length and outperforms Huffman coding with the same symbol length.

References

[Ope21] OpenMP Architecture Review Board. OpenMP application program interface version 5.2, November 2021.

[Say17] Khalid Sayood. *Introduction to Data Compression, Fifth Edition*, pages 109–113. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2017.