Integrating Error Detection into Arithmetic Coding

Colin Boyd[*], John G. Cleary, Sean A. Irvine,

Ingrid Rinsma-Melchert, Ian H. Witten

Department of Computer Science

University of Waikato

Hamilton

New Zealand

October 27, 1995

Abstract

A technique to implement error detection as part of the arithmetic coding process is described. Heuristic arguments are given to show that a small amount of extra redundancy can be very effective in detecting errors very quickly, and practical tests confirm this prediction.

[*]On leave from the School of Engineering, University of Manchester, UK. Funded by UK's Engineering and Physical Sciences Research Council (grant GR/J98790).

1 Introduction

Arithmetic coding for data compression has gained widespread acceptance as the right method for optimum compression when used with a suitable source model [1]. However, the price paid for high performance is extreme vulnerability to any errors that occur in the compressed text. Unlike Huffman coding, which recovers quickly from errors [3], a single inverted bit in arithmetically compressed text corrupts all subsequent output. It thus seems a natural idea to augment arithmetic coding with some form of error control.

The most obvious reason for wanting the source coder to act also as a channel encoder is that software, hardware, or computation time can be saved by having a single coding engine that performs both functions. But there are other arguments in favor of using arithmetic coding for error control. One is that the amount of redundancy included in encoded messages can be controlled as a single tuneable parameter of the coding process, and, if necessary, varied adaptively to accommodate prevailing channel conditions. Another is that error checking can take place continuously as each input bit is processed, so that errors are located quickly. Furthermore, an error's location can be pinned down to a small interval, reducing the amount of material that requires retransmission.

2 The Basic Idea

The essence of arithmetic coding is to partition the coding space according to the probabilities of the next source symbol, the probabilities being obtained from a separate model. Redundancy is introduced by adjusting the coding space so that some parts are never used by the encoder. During decoding, if the number defined by the received encoded string ever enters the forbidden region, a communication error must have occurred. In order to adjust the coding space, at regular points in the encoding procedure the current coding interval is reduced by a certain proportion, which we call the *reduction factor*. On decoding, the same reduction process is performed. The number defined by the received string is checked after each reduction to see if it lies within the reduced interval. If not, an error has occurred. Redundancy is controlled by varying the reduction factor.

An alternative method of adding redundancy is to use an extra model with only two symbols, encoding one of them periodically and signaling an error if the other is ever decoded. However, the chosen method has some advantages in implementation efficiency and control of redundancy.

When arithmetic coding is implemented in practice, both encoder and decoder maintain a *current interval* within which all possible values of the encoded string lie. So long as no error occurs, the encoder and decoder have identical copies of the current interval after each symbol has been processed. The current interval, defined by its length δ and lower bound l, undergoes a reduction as each new symbol is processed. In order to use finite arithmetic (and also to allow incremental encoding and decoding), it is then expanded and the leftmost bits are passed out and transmitted by the encoder (or discarded by the decoder). Expansion takes the form of doubling the interval, after subtracting an appropriate offset. Since each doubling denotes the processing of a single output bit, this is the ideal point at which to introduce redundancy. Table 1 summarizes the procedure.

How likely it is that errors will be detected, and how long will it be before they are? It turns out that the error detection behavior depends on the details of the model used. However, we give a stochastic account of the expected behavior that turns out to be very well supported by experimental results.

Consider what happens when an error occurs in a string being decoded. It results in the wrong symbol being chosen by the decoder as the inverted bit is moved through the decoding register. (If not, the error has no effect at all.) This means that the new current interval is incorrect. We regard it as a random interval since the bit in error can be of any significance. Decoding can continue, but the input string will define a new random symbol with respect to this random coding interval, and so the process continues. However, there is now a chance that the string being decoded defines a point which is outside the current interval once it is reduced. We assume that this is a random event with probability $1 - R$.

We are interested in the expected number of bits that are processed before the string departs from the current interval, causing the error to be detected. If the reduction is performed once for each bit that is processed, this waiting time is the expected number of trials before a random event of probability $1 - R$ occurs. This is distributed according to a geometric distribution whose mean and standard deviation can easily be calculated.

3 Implementation Issues

Choosing the reduction factor. The reduction factor R determines the amount of redundancy in the output, and also controls how quickly, on average, errors will be detected. Consider first how much the output is increased by multiplying the interval by R for each bit produced. If the current interval has length δ, the number of bits required to represent it is $I(\delta) = -\log_2(\delta)$. After the interval is reduced, $I(R\delta) = I(R) + I(\delta)$ bits are needed to represent it.

For a typical implementation R will be close to one and so the extra information per bit is small. Writing $1/R = 1 + r$, the approximation $\ln(1 + x) \approx x$ yields $I(R) \approx r/\ln 2$. Thus to add $x\%$ redundancy, choose $R = 1/(1+0.00693x)$. For $x = 1\%$ this evaluates to $R = 0.9931$; for $x = 2\%$, $R = 0.9863$. The predicted mean time before the error is detected is $1/(1 - R)$; 145 bits for 1% redundancy and half that number for 2%. The values chosen in a real implementation will obviously depend on the error characteristics of the channel, but we regard these as reasonable examples.

Guaranteeing detection of single errors. In many applications, errors are expected to be independent rare events, in which case the most common kind of error is an isolated single-bit inversion. In the scheme outlined above there is no guarantee that this will be detected, because there is always a small chance that the disturbed current interval will reunite with the correct one. This deficiency can be rectified by feeding the output bits forward using a 1-bit register, transmitting the usual output bit exclusive OR'd with the previous one. The input is fed back at the decoder to undo the effect. Then, single bit errors are propagated indefinitely. This means that even if the erroneous current interval does reunite with its correct value, the decoder input will be complemented. The error will eventually be found with probability that increases asymptotically to one with the length of input. This procedure also guarantees that error bursts of odd length will be detected: even-length bursts will be susceptible to the problem described above.

This idea may be exploited further by using a register with more memory. A register which simply records the previous n bits and feeds forward the oldest one may be used during encoding. By feeding back from such a register during decoding, any error pattern of length less than n would always be remembered by the register and so guarantee that the error is detected eventually. Additional properties may be obtained by including feedforward from other positions.

Protecting the end-of-file marker. Because it is usually necessary to wait for some bits to be processed before an error is detected, the last few bits of any compressed file are particularly vulnerable to undetected errors. A special EOF symbol is usually the last symbol processed during encoding, and many errors can be caught by checking that the last symbol decoded when all input bits are processed is indeed EOF. The probability of this symbol is implementation-dependent: it may be varied adaptively. Since coding terminates when it is encountered, an error may cause EOF to be recognized prematurely, truncating the decoded file with no apparent rrors. Therefore a special check symbol, with very small probability, is encoded after EOF. For example, a probability of 2^{-16} results in an additional overhead of 2 bytes. On decoding, an error is reported if the check symbol is not decoded immediately following EOF.

4 Empirical Tests

Experiments were conducted using a publicly available implementation of arithmetic coding[1] [2]. To incorporate the basic checking method, just two lines of code were added to the encoder and three to the decoder. Further simple changes were required to include a check symbol for detecting early EOF symbols, and a 1-bit feedback to guarantee detection of single-bit errors. We used the simple adaptive model distributed with the software, which gives only modest compression; but it is anticipated that similar results would be obtained with any other model.

The test input is a text file of *Alice in Wonderland*, whose 148,476 bytes reduce to 85,254 when encoded with the adaptive model. Using the value $R = 0.98633$, the output length increased by 2.04% to 85,976 bytes. Single-bit errors were introduced into the error-protected compressed file in 25,000 different bit positions. Every error was detected eventually by the decoder.

The upper part of Table 2 compares the observed mean and standard deviation with that predicted by the geometric distribution. Agreement is quite close. Moreover, the number of bits pro-

[1] A copy of the modified C programs may be obtained by anonymous ftp from `ftp.cs.waikato.ac.nz` in the file `pub/compression/error_coding.tar.gz`.

Encoding	Decoding
1. Define the reduction factor R. Its value may be chosen either to fix the percentage output redundancy, or to produce a given mean number of bits before an error is detected. 2. Whenever the current interval is doubled, replace its value by the interval with the same lower bound l but of length $R\delta$.	1. Define R as for encoding and maintain the current interval in just the same way. 2. After the current interval has been doubled and adjusted using R, check that the point defined by the received string lies inside the current interval. If not, an error has occurred.

Table 1: Encoding and Decoding Procedures

		Mean	**Standard Deviation**
2% redundancy	Predicted	73.2	72.65
	Observed	79.2	70.26
1% redundancy	Predicted	144.9	144.4
	Observed	164.1	156.3

Table 2: Number of bits processed before an error is detected

cessed before the error is detected was counted from when the error first entered the decoding register, so there is no opportunity for it to have an effect until the next decoded symbol is calculated using the register. How long this will take depends on the model, but on average it might be around half the average log of the symbol probabilities. This adjustment further improves the agreement. The experiment was repeated using $R = 0.9931$, which produced an output of 85,082 bytes or 0.998% redundancy. The result, calculated using 20,000 trials, is also shown in Table 2. Finally, experiments were conducted with no added redundancy at all. Single-bit errors were detected with a mean waiting time of around 2^{16} bits or 8 Kbytes. While unacceptable for most purposes, this is accomplished by a simple check during decoding without any change to the encoding procedure: in effect, it is error detection for free and could be a useful safety net on long files.

No attempt has been made to minimize the computational overhead incurred by adding error detection; results from our implementation indicate that it adds about 20% to encoding and 25% to decoding time. It could be greatly reduced by applying the interval reduction process only periodically. For example, including a check after every tenth bit would reduce overhead to 2% or so, and the only effect on error performance would be up to ten further bits before an error is detected.

5 Conclusion

Error detection can be incorporated into arithmetic coding in a way that is effective and has very little impact upon performance. The probability of detecting an error is similar to that of conventional techniques such as cyclic redundancy checks. When correcting errors in a communications (as opposed to a storage) environment, the new method improves significantly upon standard checksum detection because it supplies information about the position of errors. For example, with 2% redundancy, retransmitting only 320 bits on detection of an error will be sufficient to correct it about 99% of the time.

References

[1] T.C. Bell, J.G. Cleary and I.H. Witten, *Text Compression*. Prentice Hall, 1990.

[2] I.H. Witten, J.G. Cleary and R. Neal, “Arithmetic coding for data compression,” *Communications of the ACM*, *6*, June 1987, pp. 520–540.

[3] D.A. Lelewer and D.S. Hirschberg, “Data compression,” *ACM Computing Surveys*, *3*, September 1987, pp. 261–296.