

# Adaptive Heuristic Search for Soft-Input Soft-Output Decoding of Arithmetic Codes

Yali Liu<sup>1</sup>, Jiangtao Wen<sup>2</sup>

<sup>1</sup> Dept. of Electronics Engineering  
Tsinghua University  
Beijing 100084, P.R. China  
E-mail: [liuy102@mails.tsinghua.edu.cn](mailto:liuy102@mails.tsinghua.edu.cn)

<sup>2</sup> Morphbius Technology Inc.  
Ottawa, Canada K2J 3S7  
E-mail: [gwen@morphbius.com](mailto:gwen@morphbius.com)

## Abstract:

*Arithmetic coding for data compression has gained widespread acceptance for optimum compression when used in a suitable model. The "traditional" hard bit based decoding suffers from severe loss of data when errors are present in the received bitstream. In this paper, heuristic-search based algorithms are introduced for the decoding of arithmetic coded bitstreams. They utilize both channel "soft" bit reliability information and synchronization and number of symbols information to identify the most likely sequence of source symbols received.*

**Key words:** heuristic search, arithmetic coding, soft-decoding

## 1. Introduction

Recent research has demonstrated that by utilizing channel bit reliability information and synchronization and knowledge of the number of source symbols contained in a packet, one can significantly improve the performance of the decoding of bitstreams containing variable length codewords (VLCs) in the presence of transmission errors. In [1] and [2], a recursive dynamic programming based "soft" decoding algorithm for VLCs that efficiently searches the state space composed of valid codeword concatenations that produce the correct packet size in bits.

Yet another important entropy coding method that has received wide application is arithmetic codes ([3], [4]). It generally achieves better compression efficiency than Huffman codes. One of the first practical arithmetic coder was proposed in [3]. Similar to the case for variable length codes, a single bit error in the output bitstream of an arithmetic encoder may cause the decoder to lose synchronization with the encoder and result in severe error propagation and information loss. Unlike variable length codes, whose decoder may regain synchronization after a single error has occurred ([5]), errors in corrupted arithmetic coded bitstream will usually cause loss of synchronization forever.

In addition, as opposed to variable length codes that map each source symbol to a predetermined set of bits, the output from an arithmetic encoder cannot be determined by the input symbol alone, because there is no one-to-one mapping between the input symbols at the progressive output of bits from the encoder, which exhibits extremely long

memory. Therefore, explicitly building a data structure storing all possible states for all possible packets of the same number of symbols and length in bits would be impractical.

The paper is organized as follows: Section 2 contains a detailed description of two decoding algorithms based on optimal search. The first algorithm guarantees finding the “true” optimal solution but requires more space and is more time consuming. The second, simplified algorithm does not always return the optimal solution, but significantly reduces the amount of storage and calculations. Both algorithms easily out-perform the traditional “hard”-bit based arithmetic decoder. In Section 3, we introduce an algorithm utilizing embedded error detection in the arithmetic encoding and decoding process so as to further reduce the complexity of soft decoding by estimating the portion of a received packet that necessitates soft decoding. Simulation results of our proposed decoders for Gaussian white additive noise channels with various SNRs are given in Section 4. Finally Section 5 contains a summary and conclusions.

## 2. Optimal and Heuristic Search Based AC Decoding

### 2.1 Optimal Search Based Decoding Description

Our soft input soft output decoding is based on the Optimal Search. In our algorithm, each “node” in the algorithm corresponds to a particular combination of  $n$ ,  $l$ ,  $c_{n,l}$ ,  $err$ ,  $low$ ,  $high$  and  $Bits2Follow$  values. Here,  $n$  and  $l$  represent the number of symbols and bits already examined by the soft decoder.  $low$ ,  $high$  and  $Bits2Follow$  represent the current state of the arithmetic coder, which together with the next input symbol uniquely decides the subsequent output bitstream.  $c_{n,l}$  is the last codeword of the “optimal” source symbol sequence (relative to the received packet up to the  $l$ -th bit) with  $n$  symbols,  $l$  bits, as well as an arithmetic encoder with  $low$ ,  $high$  and  $Bits2Follow$  values after encoding  $c_{n,l}$ .

Pseudo code for the decoding is given below. The reader is respectfully referred to [7] for a background on **OptimalSearch** and proof of its optimality.

#### **Optimal Search Based Decoding ()**

1.  $Open = \text{CreatEmptyList}()$ ;
2.  $Closed = \text{CreatEmptyList}()$ ;
3. **AddToList** ( $Open$ ,  $Initial\ Node$ );
4. Loop:
5.   if **ListIsEmpty** ( $Open$ ) exit ( $FAIL$ );
6.    $node = \text{GetFirstFromList}$  ( $Open$ );
7.   if **IsSolution** ( $node$ ) exit ( $SUCCESS$ );
8.   **RemoveFromList** ( $node$ ,  $Open$ );
9.   **AddToList** ( $Closed$ ,  $node$ );
10. for (all  $symbol$  in source alphabet) {
11.      $bits2follow = node.Bits2Follow$ ;
12.      $low = node.low$ ;    $high = node.high$ ;
13.      $range = node.high - node.low$ ;

```

14.   GetNewHigh(high, low);
15.   GetNewLow(high, low);
16.   ACEncode(symbol, high, low);
17.   dist = CompareBitPlusFollow();
18.   newNode = CreatNewNode();
19.   if (newNode in neither Close nor Open) {
20.       AddToList(Open, newNode);
21.        $g(\text{node}_{\text{symbol}}) = g(\text{node}) + \text{dist}$ ;
22.       Prev(nodesymbol) = node;
23.   }
24.   else if (newNode in Open) {
25.        $g_{\text{newNode}}(\text{symbol}) = g(\text{node}) + \text{dist}$ ;
26.       if ( $g_{\text{newNode}}(\text{symbol}) < g(\text{newNode})$ ) {
27.           Prev(newNode) = node;
28.            $g(\text{newNode}) = g_{\text{newNode}}(\text{symbol})$ ;
29.       }
30.   }
31.   Find smallest g-value in Open;
32.   Goto Loop;
(NumSymbols is the number of symbols in the source alphabet)

```

#### **CompareBitPlusFollow**(*bit, l*)

```

1.  {
2.      double d;
3.       $d = \text{cm}(\text{bit}, \text{Received}[l])$ ;
4.      for ( $i=l+1; i \leq l+\text{BitsToFollow}; i++$ ) {
5.           $d += \text{cm}(!\text{bit}, \text{Received}[i])$ ;
6.      }
7.      return d;
8.  }

```

( $\text{cm}(\text{bit}, x)$  is a closeness metric for a binary *bit* and a possibly floating point received value *x*)

After the decoder exists from **Optimal\_Search\_Based\_Decoding**(), the decoder needs to trace back the optimal route identified so as to reconstruct the optimal sequence of source symbols. For this purpose, some additional data needs to be stored during the search. For sake of clarity, we did not include these auxiliary data structures in the description of the search algorithm and associated main data structures.

## **2.2 Data Structures**

The processing between lines 10-30 in the above pseudo code is called “expanding” *node*. *Closed* and *Open* in the above algorithm saves the nodes that are already expanded and nodes to be expanded. That intelligently designed data structures for storage of *Closed* and *Open* are required so as to achieve a good trade-off between storage and computational efficiency, for it would need to record all information contained in each node, namely  $n, l, c_{n,b}, err, low, high$  and *Bits2Follow*.

In our simulations, we use an array of linked list for both *Closed* and *Open*. For *Open*, we note that because optimal search is not exhaustive in the sense that not all nodes in *Open* need to be expanded before the algorithm terminates with a solution to the search problem, all  $[0, L]$  will not be encountered during the search. On the other hand, all of the  $N$  possible source symbols need to be tested whenever a node is expanded. We define *Open* as an array of  $N$  linked lists of the data structure *Node*:

For *Closed*, an array of  $n$  linked lists is used, where  $n$  is the number of symbols in the packet to be decoded. Each entry in the linked list is of type *flag*, which contains information such as  $l, low, high, Bits2Follow$  so to identify nodes. In addition, an array of bytes is included, with the  $i$ -th bit of the  $j$ -th byte in the byte array corresponding to the  $(i*8)+j$ -th symbol in the source alphabet. When this bit is 1, it designates that the  $node = \{n-1, l, low, high, Bits2Follow\}$  has already been expanded by the decoder, and the last source symbol of the optimal path to this node found so far is the  $(i*8)+j$ -th symbol in the source alphabet.

### 2.3 Using Heuristic in the Decoding process

Although the algorithm in 3.1 was defined so as to simplify the search decoding process, the number of expansions could increase dramatically when *Bits2Follow* becomes non-zero. This is because the algorithm chooses the next candidate for expansion based on the closeness measure of the nodes from the partial packet that has been already processed. For some nodes however, expansion will only result in the increase of *Bits2Follow* without any output bits for a number of source symbols for the for-statement of line 9. As a result, the closeness measure does not change from its value prior to expansion for these symbols and the decoder will have a hard time choosing a good candidate for the next expansion using the closeness measure as a metric.

To solve this problem, we note that when *Bits2Follow* is not zero, whatever bit output by the AC will be followed by *Bits2Follow* opposite bits. Therefore, although we don't yet know how the closeness measure changes as a result of the expansion, we could use the knowledge of *Bits2Follow* to obtain a *lower bound* of the change in closeness measure.

As an example, suppose our closeness measure is the Hamming distance and that after expansion *node* with *symbol*,  $Bits2Follow=2$ . This means that the subsequent output from the AC should either be 011...or 100... . If the received bits at the corresponding positions are 111, we know that the Hamming distance after expanding *node* with *symbol* is *at least* 1 (corresponding an output from the AC of “0”, which results in an output of “011” ). We denote this low bound obtained by examining *Bits2Follow* and the received

bits by  $tmpCM$ , and use  $g'(node)=g(node)+tmpCM-n-l(-1)$  as the criteria for deciding the next node in *Open* to be expanded. The inclusion of  $n$ , and  $l$  values here is to give precedence to nodes with larger  $n$ ,  $l$  values so as to give precedence to nodes that are near “completion” during the expansion process to further improve decoding speed. (the operation in () will be performed when *Bits2Follow* is none zero)

As  $g'$  is not determined solely by the closeness measure  $cm$ , the heuristic search based on  $g'$  in general will not result in the optimal solution with respect to the closeness measure, however, as the results in the next section show, the “heuristic” information in the definition of  $g'$  significantly improved the speed of the decoding with a reasonable degradation in performance and is therefore a good tradeoff.

### 3. Integrating Error Detection into Soft Decoding

Although “soft” decoding can achieve significantly better performance than traditional “hard” decoding, the decoding time increases rapidly with the number of symbols and noise level, even with the presence of heuristic information. One way of overcoming this problem is to integrate effective error detection into soft decoding, so that traditional, computationally efficient hard decoding can be performed as much as possible while computationally intensive soft decoding is only invoked for portions of the received packets that contain transmission errors.

#### 3.1 Error Detection with Forbidden Symbol (FS)

A well-studied means of error detection in arithmetic coding ([8], [9]) is to introduce a dummy forbidden symbol (FS)  $m$  with probability  $P_m = e$  into the source input alphabet, which is never actually transmitted. Then during the decoding process, as soon as the FS is decoded as a result of transmission errors, an “educated guess” is made about the location of the error.

Obviously, from a coding efficiency perspective, this mechanism corresponds to perturbing the source mode by a factor  $1 - e$ , thereby introducing an encoding overhead of ([9])

$$R_x = -\log_2(1 - e) \text{ bits/symbol} \quad (1)$$

[9] showed that when using hard decoding, once the FS is detected, there is a  $(1 - d)$  probability that errors occurred in the previous  $n$  bits, where

$$n = \frac{\log_2(d)}{\log_2(1 - e)} \quad (2)$$

Using the FS as a means of error detection, we can start decoding a received packet with hard decoding and record the number of bits processed for each decoded symbol. Then as soon as an FS is detected, we decide, based on a certain confidence level, where the error occurred and how many symbols decoded so far with hard decoding is therefore un-

reliable. Then we start soft decoding from the first bit position and the symbol where the hard decoding result is deemed corrupted, until the end of the packet.

### 3.2 Error Detection Based on EoS symbol

One possible problem with error detection with a forbidden symbol is that although the mechanism can provide a fair good estimate to the *bit* location of the transmission errors, the correspondence between bit positions and source symbols can often be ambiguous. This is because in arithmetic encoding, the encoding of input symbols may sometimes result only in changes of the values of *low*, *high*, and *Bits2Follow*, without any immediate output. It is desirable therefore, to find an error detection mechanism that can better pinpoint the symbols that are not reliably decoded, as opposed to bits that are not reliably transmitted.

To this end, in our simulations, we used a simple mechanism of inserting a special EoS symbol periodically (e.g. every  $k-1$  source symbols) into the source sequence of symbols to be encoded. Then when performing hard decoding, we check if the EoS symbol is present at the expected locations. If not, soft decoding will be performed from the last EoS that was recovered at expected locations. It should be noted that, during this process, the EoS symbol is encoded and decoded as any other symbol in the source alphabet, this is in contrast to partitioning the input source to small packets of  $k-1$  symbols, as in the latter case, the encoder will have to be initiated with  $low=0$ ,  $high=65535$ ,  $Bits2Follow=0$ , and terminated, thereby introducing a large overhead.

To calculate the overhead introduced by inserting the EoS for every  $k-1$  source symbols in an  $m$ -symbol packet, we assume that the size of alphabet is  $N$ , where symbol probabilities are  $p_i(x)$ . Let  $H_1$  denote the entropy of the original  $m$ -symbol packet then we have

$$H_1 = -\sum_{i=1}^N p_i(x) \log_2(p_i(x)) \quad (3)$$

Since our EoS insertion scheme corresponding to a source with the EoS having a probability of  $\frac{1}{k}$ , and the other symbols having probabilities of  $p_i(x) \left(1 - \frac{1}{k}\right) = \hat{p}_i(x)$ , average length of the  $m$ -symbol packet after EoS insertion can be estimated as

$$H_2 = \left(m + \frac{m}{k-1}\right) \cdot \left(-\sum_{i=0}^N \hat{p}_i(x) \log_2(\hat{p}_i(x)) - \frac{1}{k} \log_2 \frac{1}{k}\right) . \quad (4)$$

Therefore the redundancy introduced will be

$$R_x = H_2 - H_1 = \left(m + \frac{m}{k-1}\right) \cdot \left(-\sum_{i=0}^N \hat{p}_i(x) \log_2(\hat{p}_i(x)) - \frac{1}{k} \log_2 \frac{1}{k}\right) - m \cdot \left(-\sum_{i=1}^N p_i(x) \log_2(p_i(x))\right)$$

or

$$-\log_2 \left(1 - \frac{1}{k}\right) - \frac{1}{k-1} \log_2 \frac{1}{k} \text{ (bits/symbol)}. \quad (5)$$

Comparing (5) and (1), we see that when using the same interval length to represent the EoS or  $\mathbb{H}$ , the additional redundancy introduced by the actual EoS insertion into the encoding process is  $-\frac{1}{k-1} \log_2 \frac{1}{k}$  bits/symbol, which is cost the decoder has to pay for pinpointing erroneously decoded segments in the received packet.

#### 4. Simulation Results

To demonstrate the effectiveness of the heuristic search algorithm, we used an i.i.d. source with 4 symbols and a distribution of  $\{0.4, 0.3, 0.2, 0.1\}$  and compared the performances of the optimal soft decoder, the soft decoder with heuristic information and traditional hard decision based decoder. We assumed that the output of the encoder were transmitted as 1.0 for binary 1 and -1.0 for binary 0 over an additive white Gaussian noise (AWGN) channel, and ran simulations at different signal to noise ratios (SNRs). We used the-sum-of-the-square-errors (SSE) between the received and original values for all bits in the packet as our closeness measure. For each SNR level, we randomly generated 20 packets each containing 15 source symbols. For each such packet and SNR combination, we produced 50 noise sequences. The average packet length was about 28.65 bits. From the results in TABLE I we can see that both soft decoders easily out-performed the hard decoder, while the heuristic decoder provided a very good trade-off between complexity and performance. In the table, packet error rate is the probability of for a decoded packet to be different from the original at *any* position; symbol error rate is the probability for the symbols in the decoded packet to be different from the original at corresponding positions. Average decoding time for each packet was obtained using a Pentium 4 2.0GHz PC with 256M memory running the Windows 2000 operation system. Although the execution time is a function of many factors in addition to the inherent complexity of the algorithm involved, from the decoding time of the *same* soft decoders for various noise levels, we can see that the their complexities decrease as the noise level lowers, as one optimal path will emerge quickly and less nodes need to be expanded.

TABLE II compared the performance of the heuristic decoder and hard decoder when decoding packets of larger sizes and source with larger alphabet, we further compared their performance when decoding an i.i.d. source with 16 symbols and a distribution of  $\{0.1, 0.095, 0.090, 0.085, \dots, 0.025\}$ , the entropy of the source was about 3.9 bits/symbol. For each SNR level, we randomly generated 20 packets each containing 25 source symbols. For each such packet and SNR combination, we produced 50 noise sequences. The average packet length was about 100 bits. Due to its prohibitively high decoding time, the performance of the optimal soft decoder was not tested. As can be seen from the results, the heuristic decoding provide significantly better performance than hard decision decoding even as the packet size and source distribution become non-trivial, and that the decoding time increases fairly rapidly with the increase of packet size and noise levels.

Finally we tested the performance of the heuristic decoder with embedded error detection using the aforementioned 16-symbol source but larger packets with 45 symbols each. For each simulation, we still randomly generate 50 packets, and for each such packet and

SNR combination, we still use 50 different noise sequences. The length of the FS interval was  $1/16$ , whereas  $k=16$  for EoS insertion based error detection. The average packet length was about 193 bits. The overhead for FS based error detection was 0.09 bits/symbol, for EoS insertion based detection, it was 0.36bits/symbol. As a comparison, the entropy of the original 16-symbol source was about 3.9 bits/symbol. The results in the table showed that EoS insertion based error detection, with its larger redundancy and simplicity in isolating error corrupted symbols, achieved the best trade off between performance and speed.

## 5. Conclusions and Future Work

In this paper, we presented soft-input soft-output decoding algorithms for corrupt arithmetic bitstreams. The algorithms were based on optimal and heuristic search algorithms in artificial intelligence and provided significantly better performance than the traditional hard bits based decoder. Two different error detection schemes were also tested with soft decoding, which showed very promising combination of speed and performance for practical applications.

Possible areas of further study include soft decoding for ACs that adapts to source distributions, various heuristic knowledge based search algorithms to further reduce complexity, and etc.

### References:

- [1] J. Wen, J. Villasenor, "Utilizing soft information in decoding of variable length codes", *Proc. of IEEE 1999 Data Compression Conference*, pp. 131 - 139. Snowbird, UT, March 1999.
- [2] J. Wen, J. Villasenor, "Soft-input soft-output decoding of variable length codes", *IEEE Transactions on Communications*, Vol. 50, No. 5, pp. 689-692, May 2002.
- [3] I.H. Witten, R.M. Neal, J.G. Cleary, "Arithmetic coding for data compression", *Communications of the ACM*, Vol. 30, No.6, pp.520-540, June 1987.
- [4] G.G. Langdon, Jr., "An introduction to arithmetic coding", *IBM Journal of Research and Development*, Vol. 28, No.2, pp.135-149, March 1984.
- [5] J. Wen, J. Villasenor, "Reversible variable length codes for efficient and robust image and video coding", *Proc. of IEEE 1998 Data Compression Conference*, pp. 471 - 480. Snowbird, UT, March 1998.
- [6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, Second Edition, MIT Press, Sept. 2001.
- [7] N.J. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[8] C. Boyd, J. Cleary, S. Irvine, I. Rinsma-Melchert, and I. Witten, "Integrating error detection into arithmetic coding," *IEEE Trans. Comm.*, vol. 45, no. 1, pp. 1–3, Jan. 1997.

[9] J. Chou and K. Ramchandran, "Arithmetic coding-based continuous error detection for efficient ARQ-based image transmission," *IEEE J. Select. Areas Comm.*, vol. 18, no. 6, pp. 861–867, June 2000.

**TABLE I**  
**Comparison of Optimal SISO and Heuristic SISO Decoding of Arithmetic Codes**

$\sigma$ (SNR)	Optimal Soft Decoding		Heuristic Soft Decoding		HD Decoding	
	PER/SER	Avg. Decoding Time(s)	PER/SER	Avg. Decoding Time(s)	PER/SER	Avg. Decoding Time (s)
0.2 (12dB)	0/ 0	0.080	0/ 0	0.055	0/ 0	0.133
0.3 (10dB)	0/ 0	0.249	1.0E3/ 2.7E4	0.052	8.0E3/ 3.3E3	0.133
0.4 (8dB)	2.7E2/ 9.5E3	14.264	6.3E2/ 2.6E2	0.053	1.6E1/ 6.5E2	0.133

**TABLE II**  
**Comparison of Heuristic SISO and Bit-Wise Hard Decision Based Decoding of Arithmetic Codes**

$\sigma$ (SNR)	Heuristic Soft Decoding		HD Decoding	
	PER/SER	Avg. Decoding Time(s)	PER/SER	Avg. Decoding Time (s)
0.2 (12dB)	0/0	0.153	0/0	0.408
0.3 (10dB)	1.1E2/5.0E3	0.161	3.6E2/2.1E1	0.304
0.4 (8dB)	3.4E1/2.1E1	7.559	4.7E1/2.6E1	0.276

**TABLE III**  
**Comparison of SISO and Bit-Wise Hard Decision Based Decoding of Arithmetic Codes**

$\alpha$ (SNR)	Heuristic Soft Decoding (eof)			HD Decoding		
	PER	SER	Avg. Decoding Time(s)	PER	SER	Avg. Decoding Time(s)
0.2 (12dB)	0	0	0.042	0	0	0.042
0.3 (10dB)	0.001	6.67e-004	0.043	0.076	0.037	0.043
0.4 (8dB)	0.064	0.033	5.699	0.68	0.39	0.043