

Binary Arithmetic Coding With Key-Based Interval Splitting

Jiangtao (Gene) Wen, Hyungjin Kim, and John D. Villasenor

Abstract—Binary arithmetic coding involves recursive partitioning the range $[0,1)$ in accordance with the relative probabilities of occurrence of the two input symbols. We describe a modification of this approach in which the overall length within the range $[0,1)$ allocated to each symbol is preserved, but the traditional assumption that a single contiguous interval is used for each symbol is removed. A key known to both the encoder and decoder is used to describe where the intervals are “split” prior to encoding each new symbol. The repeated splitting has the effect of both scrambling the intervals and altering their lengths, thereby allowing both encryption and compression to be obtained simultaneously.

Index Terms—Arithmetic codes, cryptography, data compression.

I. INTRODUCTION

ARITHMETIC coding has been developed over several decades [1] and has recently been adopted for use in compression standards, including JPEG2000 and H.264. It involves associating a sequence of symbols with a position in the range $[0,1)$, and is usually implemented recursively. Because multiple symbols are coded jointly, arithmetic coding typically enables very high coding efficiency.

Here we consider a modification of arithmetic coding that provides both compression and encryption. This is an area that appears to have received very little attention in the literature. Cleary *et al.* [2] have shown that arithmetic coding as traditionally implemented is not particularly secure. Bergen *et al.* [3] have considered the problem of inferring the underlying symbol probabilities and partitioning of the $[0,1)$ interval using observations of an arithmetic encoder output. Liu *et al.* [4] use table-based bit sequence substitutions to provide encryption during arithmetic coding. The most closely related work is that of Grangetto *et al.* [5], [6], who describe random swapping of the two intervals in a binary arithmetic coder and use this to encrypt JPEG 2000 coded images.

By contrast, we adopt an approach in which the intervals associated with each symbol, which are continuous in a traditional arithmetic coder, can be split according to a key known both to the encoder and decoder. For example, in a binary system with

two symbols **A** and **B** and $p(\mathbf{A}) = 2/3$ and $p(\mathbf{B}) = 1/3$, a traditional partitioning would represent **A** by the range $[0,2/3)$ and **B** by the range $[2/3,1)$. We remove the constraint that the intervals corresponding to each symbol be continuous and instead use a more generalized constraint that the sum of the lengths of the one or more intervals associated with each symbol be equal to its probability. With reference to the example above, if symbol **A** is represented by the combination of the intervals $[0,1/3)$ and $[2/3,1)$ and symbol **B** by $[1/3,2/3)$, the fundamental association in arithmetic coding between symbol probability and length in the range $[0,1)$ is obviously preserved. This can be viewed as a generalization of the method in [5] and [6], in the sense that it results in a coder having intervals that have been both randomly shuffled **and** split.

Arithmetic coding can be viewed as an extension of Shannon–Fano–Elias coding and associates a concatenation of N symbols x^N with prefix-free codewords of length no greater than $l(x^N) = \lceil -\log p(x^N) \rceil + 1$ bits [7]. Alternatively, if the prefix-free restriction is removed, codewords of length $\lceil -\log p(x^N) \rceil$ can be used. For a given sequence length N and input probability distribution with cdf $F(i)$, the recursive partitioning of $[0,1)$ leads to a set of intervals $[F(i-1), F(i))$, $1 \leq i \leq 2^N$ and their associated midpoints $\bar{F}(i)$. The representation of $\bar{F}(i)$ is truncated to $l(x^N)$ bits to give $\lfloor \bar{F}(i) \rfloor_{l(x^N)}$, which is guaranteed to lie in the appropriate interval.

In this letter, we apply the above principles to symbol intervals that have been split into two disjoint subintervals. Coding of an input sequence is accomplished by finding a number representing a location within one of the two subintervals. Since the longer of the subintervals must be at least half as long as the pre-split interval, a prefix-free representation will require at most $l(x^N) = \lceil -\log p(x^N) \rceil + 2$ bits (or $l(x^N) = \lceil -\log p(x^N) \rceil + 1$ when the prefix-free restriction is removed). In many cases, a representation shorter than these bounds is available within the longer subinterval. Alternatively and less commonly, the shortest representation will lie in the shorter subinterval. The encoder has the flexibility to pick the shortest valid codeword from either subinterval. Thus, as the performance bound is 1 bit per N -symbol sequence larger than traditional arithmetic coding, the efficiency penalty in relative terms becomes vanishingly small as N increases. Furthermore, as discussed in Section III, in practice, the bound is quite loose, and the code lengths with interval splitting tend to be approximately 0.5 bits longer than code lengths produced using traditional arithmetic coding. In return for this overhead, significant secrecy is obtained. While the concept of interval splitting can be applied to a source alphabet with any size, for simplicity, the remainder of the discussion focuses on the binary case.

Manuscript received August 23, 2005; revised September 26, 2005. This work was supported in part by the Office of Naval Research and in part by the National Science Foundation. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Hongbin Li.

J. G. Wen is with Mobilygen Corporation, Santa Clara, CA 95054 USA (e-mail: gwen@mobilygen.com).

H. Kim and J. D. Villasenor are with the Electrical Engineering Department, University of California, Los Angeles, CA 90095 USA (e-mail: hjkimnov@ee.ucla.edu; villa@icsl.ucla.edu).

Digital Object Identifier 10.1109/LSP.2005.861589

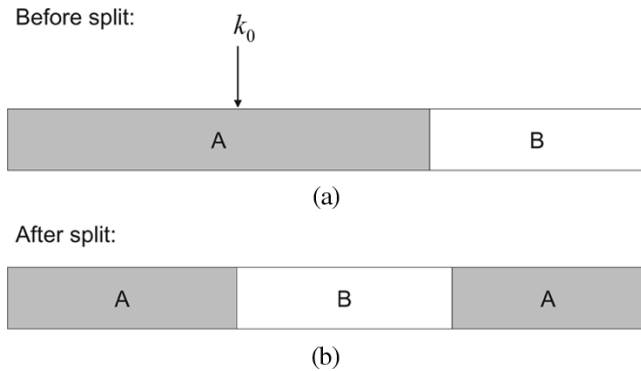


Fig. 1. Illustration of interval splitting.

II. INTERVAL SPLITTING

We first illustrate the concept of interval splitting for the case where only one input symbol is coded. Fig. 1(a) shows a traditional partitioning of $[0,1]$ using an example where $p(\mathbf{A}) = 2/3$ and $p(\mathbf{B}) = 1/3$. A key k_0 can be used to identify where the interval corresponding to symbol \mathbf{A} is to be split. The split causes the portion of the \mathbf{A} interval to the right of the key to be moved to the right of the \mathbf{B} interval, as shown in Fig. 1(b). Because the \mathbf{B} interval has not been split and therefore is still contiguous, if the input symbol is \mathbf{B} , then it can be represented in a prefix-free manner using the $\lceil -\log\{p(\mathbf{B})\} \rceil + 1$ bit truncation of the midpoint of the interval, just as it occurs in traditional arithmetic coding. If the input symbol is \mathbf{A} , then it is necessary to find a number representing a position in either of the two \mathbf{A} subintervals. As discussed above, this can be expressed using at most $\lceil -\log\{p(\mathbf{A})\} \rceil + 2$ bits and, in some cases, $\lceil -\log\{p(\mathbf{A})\} \rceil + 1$ bits.

The partitioning becomes more constrained when multiple symbols are considered. The key becomes a vector $k = (k_0, \dots, k_{N-1})$, where N is the length of the input symbol string. Also, splitting of the intervals can no longer be performed at arbitrary locations. In keeping with the recursion approach to arithmetic coding, the regions associated with either \mathbf{A} or \mathbf{B} when encoding the first symbol are partitioned again when a second symbol is encoded. With reference to Fig. 1(b), the interval for \mathbf{B} is contiguous, so for the second symbol, it is partitioned in the traditional manner, as shown in Fig. 2(a). Symbol \mathbf{A} , however, is represented as the union of two disjoint subintervals, so the partitioning associated with a second symbol will introduce a boundary into one of the two subintervals (in this example, the right subinterval), as shown in Fig. 2(a). Next, the second element of the key vector k_1 is applied. The key location is indicated in Fig. 2(a) and appears twice: once, respectively, for each of the cases where the first symbol encoded was \mathbf{A} or \mathbf{B} . The intervals after splitting at k_1 are shown in Fig. 2(b). As in traditional arithmetic coding, the overall length of the intervals associated with each of the four possible pairs of input symbols is equal to the input probability; the difference is that the subintervals are no longer contiguous. Also, note that in Fig. 2(b), each symbol pair is represented by at most two distinct intervals.

As noted above, there are constraints on key choice. Fig. 2(c) illustrates the result of an inappropriate value for k_1 , which

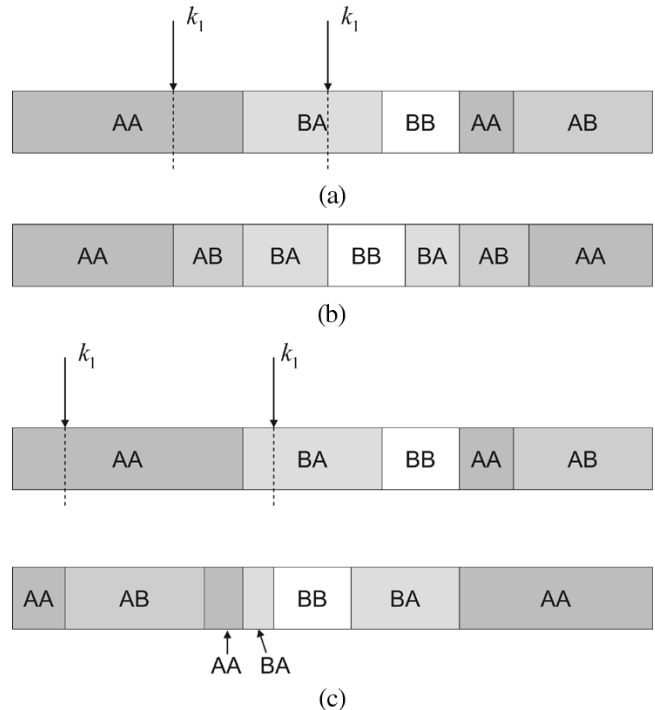


Fig. 2. Illustration of interval splitting with two symbols.

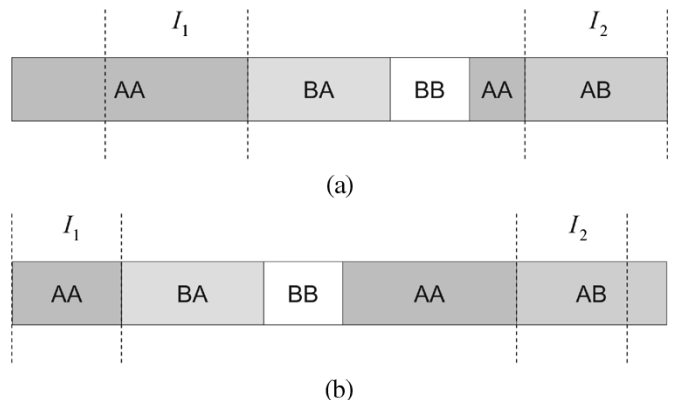


Fig. 3. Illustration of constraints on interval splitting for two symbols.

creates three subintervals associated with symbol pair \mathbf{AA} . Choosing a key as indicated in Fig. 2(c) would remove the ability to bound the efficiency of the system as discussed previously. In order to avoid the creation of three subintervals and the associated potential efficiency loss, constraints shown in Fig. 3 are imposed. Fig. 3(a) shows the constraints when, as is the case in Fig. 2(a), the interval \mathbf{AB} is smaller than the left subinterval. The key location must lie either within the interval I_2 that spans the full \mathbf{AB} interval or within interval I_1 , which is equal in size to I_2 and shares the same maximum value as the left \mathbf{AA} subinterval. Fig. 3(b) shows the case in which the left \mathbf{AA} subinterval is smaller than the \mathbf{AB} interval. The key must lie within either I_1 or I_2 , as shown in the figure. Such restrictions will occur each time a new symbol is encoded. In general, as the relative symbol probabilities and key locations vary, the positions and sizes of the I_1 and I_2 intervals will vary as well, but $\text{length}(I_1) = \text{length}(I_2)$ will always hold. For convenience, keys are expressed in a normalized manner, with

TABLE I
COMPARISON OF CODE LENGTHS AS A FUNCTION OF SEQUENCE LENGTH N

Symbol Prob.	N	$H \times N$	Traditional	Interval Splitting	Efficiency penalty in %
$p(\mathbf{A})=2/3$ $H=\text{Entropy}=.9183$	10	9.183	10.21	$10.87 \pm .06$	6.38 ± 0.54
	100	91.830	92.93	$93.46 \pm .07$	0.56 ± 0.08
	1000	918.296	919.10	$919.71 \pm .08$	0.066 ± 0.009
	10000	9182.958	9183.23	$9183.84 \pm .07$	0.0067 ± 0.0008
$p(\mathbf{A})=6/7$ $H=\text{Entropy}=.5917$	10	5.917	6.91	$7.60 \pm .16$	10.0 ± 2.4
	100	59.167	60.34	$60.84 \pm .03$	0.83 ± 0.05
	1000	591.673	592.94	$593.44 \pm .05$	0.084 ± 0.004
	10000	5916.728	5917.62	$5918.15 \pm .03$	0.0090 ± 0.0006

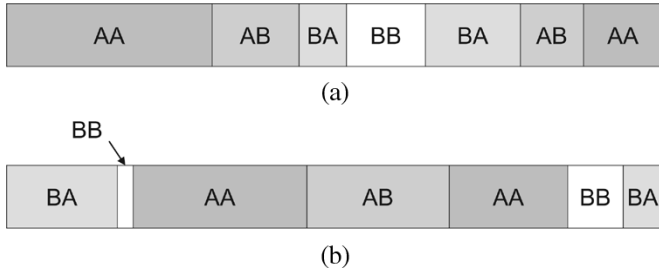


Fig. 4. Interval ordering examples after encoding of two symbols using two keys. (a) $(k_0, k_1) = (.45, .23)$. (b) $(k_0, k_1) = (.85, .4)$.

the union of I_1 or I_2 mapped to the range $[0,1)$. A key value lying between 0 and 0.5 specifies a location in the I_1 interval, and key values greater than 0.5 denote positions within I_2 .

There is an enormous diversity of interval ordering enabled by the use of keys. Fig. 4 shows two examples, both based on coding of two symbols from a source with $p(\mathbf{A}) = 2/3$ and $p(\mathbf{B}) = 1/3$. Fig. 4(a) and (b) show the ordering associated with the keys $(k_0, k_1) = (.45, .23)$ and $(.85, .4)$, respectively.

To quantitatively measure the diversity created by the set of possible output orderings, we associate each input sequence with a binary number between 0 and $2^N - 1$ using a lexicographic mapping in which \mathbf{A} and \mathbf{B} are represented by 0 and 1, respectively. In a traditional arithmetic coder, the $[0,1)$ range contains 2^N intervals also arranged in lexicographic order, and binary input symbol sequence m is represented using a number specifying a location in the m th interval. When splitting is used, there will be $2^{N+1} - 1$ intervals. This is because all but one of the possible symbol strings will be represented by two intervals. One of these intervals is greater than or equal in length to the other, but the codeword that will be chosen to represent the symbol string could be drawn from either interval. One string [BB in Fig. 4(a) and AB in Fig. 4(b)] will be represented by only one interval. Thus, although there are $2^{N+1} - 1$ total intervals, only 2^N of these will actually be used, with one for each of the possible 2^N inputs. Input sequence m will be represented using the $y(m)$ th such interval. Clearly, the output ordering after splitting is no longer lexicographic.

A diversity measure can be constructed by taking the average of the distance, measured using the number of intervening “longer” subintervals, associated with lexicographically adjacent input sequences

$$R = \frac{1}{2^N - 1} \sum_{m=1}^{2^N - 1} |y(m) - y(m-1)|.$$

For traditional arithmetic coding, the analogous measure gives $R = 1$, since lexicographically adjacent input sequences are represented using intervals that are themselves adjacent. When arbitrary permutations are permitted, it can be shown that the maximum value that R can take is $R_{\max}(N) = (2^{2^N - 1} - 1)/(2^N - 1)$, which quickly approaches 2^{N-1} as N increases. Interval splitting does not permit all possible permutations because the resulting interval orderings are symmetric, as can be seen in Fig. 4. However, it still enables a maximum R of 2^{N-1} , which is extremely close to R_{\max} . The ability to achieve diversities approaching R_{\max} has been confirmed in simulations using randomly selected keys.

III. EFFICIENCY, KEY PRECISION, AND SECRECY

Table I shows the results of applying interval splitting for input sequences with length $N = 10, 100, 1000$, and 10000 symbols and allows comparison in absolute and relative terms with traditional arithmetic coding. The upper half of the table considers the case where $p(\mathbf{A}) = 2/3$, and the lower half of the table considers the case where $p(\mathbf{A}) = 6/7$. The code lengths shown in the table are averages based on simulations using 1000 random sequence realizations. Since the exact length of the output when interval splitting is used depends not only on the input data but also on the specific sequence of split locations used, the column labeled “Interval Splitting” gives the mean and standard deviation of the code lengths based on a large number of simulations using random seeds for splitting. The right-most column in the table gives the corresponding mean and standard deviation for the efficiency penalties.

The results in the table confirm that the efficiency penalty in percentage terms becomes smaller with increasing N , falling to approximately 0.6% for $N = 100$ and to 0.007% for $N = 10\,000$ when $p(\mathbf{A}) = 2/3$. In absolute terms, the efficiency penalty with respect to traditional arithmetic coding is well under 1 bit and typically closer to 0.5 bits. Measured with respect to the bounds, the penalty is even lower. Traditional arithmetic coding guarantees prefix-free code lengths no longer than $\lceil -\log p(x^N) \rceil + 1$ bits, while with interval splitting, the bound is $\lceil -\log p(x^N) \rceil + 2$ bits. In practice, however, the bound for interval splitting is very loose, and the average prefix-free code lengths typically range from 0.1 to 0.3 bits longer (for the entire sequence of N symbols) than the expected value of $\lceil -\log p(x^N) \rceil + 1$ (the length differences in the table are larger than 0.3 bits because traditional arithmetic coding also typically requires less than the bound, though not by as large a margin as when interval splitting is used). In addition, it also should be noted that the modest efficiency penalty is the cost of obtaining secrecy. It arises because of the disjoint nature of the intervals, which in turn contributes to the secrecy since it introduces a perturbation on the traditionally more direct association between symbol string probability and the length of the resulting representation.

With regard to implementation complexity, a key-based interval splitting arithmetic coder can be implemented using techniques similar to those used in traditional arithmetic coding and can benefit from the same optimizations for speed, finite precision, etc. The main difference lies in the doubling of the number of intervals, which doubles the memory requirement because the upper and lower limits of two intervals must be maintained. In addition, renormalization must utilize both the interval length and the key, introducing an additional multiplications, though as with traditional arithmetic coding, in some cases, faster algorithms that replace the multiplications with simpler operations can be introduced.

Another important issue is the key itself, which must of course be known or communicated to the decoder. The number of potential split locations—and therefore the level of secrecy—are determined in part by the precision of the key. The key can be used to directly identify split locations, or it can reference locations in a table known to both the encoder and decoder. When the table approach is used, even a 1-bit key used to select between two possible split locations with each new symbol can provide substantial secrecy since the number of potential split orderings in a length- N sequence grows as 2^N . From the standpoint of allowing diverse interval orderings, increasing the number of potential split locations gives more possibilities, though we have found that for binary arithmetic coding, having more than four split locations (e.g., keys of more than 2 bits per symbol) offers little increased benefit.

From a coding efficiency standpoint, devoting 2 bits/symbol to a key may seem high, especially since the entropy of the symbol stream itself is under 1 bit/symbol. However, there are many approaches to dramatically reduce or eliminate the key “overhead.” These approaches can be used alone or in combination. First, a key can be reused for many sets of sequences, or equivalently, it can be used periodically in a single, long symbol

sequence. Cyclic reuse of keys is a commonly used method in cryptography. Second, the resolution of the key can be reduced. For example, even a 1 bit/symbol key creates a large diversity of possible interval orderings once the number of encoded symbols is nontrivial. Third, keys can be fully or partially self-generated, based on specific bit sequences created during the encoding process for a particular set of input symbols, on the input sequence itself, or on other information available at both the encoder and decoder, as in [5] and [6]. In the above cases, both the encoder and decoder need to have advanced knowledge of the key generation algorithm, but explicit transmission of the key itself is avoided. A number of other variations are also possible, including splitting only for a subset of symbols, varying the specific input symbol interval (\mathbf{A} or \mathbf{B}) subject to the split, conditional splitting, etc.

Like other encryption methods such as the Advanced Encryption Standard (AES), the coder presented here can be attacked using brute force if the attacker has possession of an input, corresponding output, knowledge of the exact or approximate algorithmic approach, and the ability to try enormous numbers of key possibilities. However, as with AES, the key-based arithmetic coder can be made robust to attack simply by using a sufficiently large key.

IV. CONCLUSION

We have presented a modification of arithmetic coding that uses key-based interval splitting to simultaneously enable data compression and encryption. We have shown that even when intervals in an arithmetic coder are split, the code length increase relative to traditional arithmetic coding is bounded to less than 1 bit per N -symbol sequence, and in practice, the increase is often approximately 0.5 bits per N -symbol sequence. In percentage terms, this efficiency penalty becomes negligibly small as N increases. The splitting produces encryption, the level of which is a function of the specific attributes of the key and the encoded sequence. While we have focused on the static binary case for simplicity, the methods presented here can also be applied to M -ary and/or adaptive arithmetic coding.

REFERENCES

- [1] G. Langdon and J. Rissanen, “Compression of black-white images with arithmetic coding,” *IEEE Trans. Commun.*, vol. COM-29, no. 6, pp. 858–867, Jun. 1981.
- [2] J. Cleary, S. Irvine, and I. Rinsma-Melchert, “On the insecurity of arithmetic coding,” *Comput. Security*, vol. 14, no. 2, pp. 167–180, 1995.
- [3] H. Bergen and J. Hogan, “A chosen plaintext attack on an adaptive arithmetic coding algorithm,” *Comput. Security*, vol. 12, no. 2, pp. 157–167, Mar. 1993.
- [4] X. Liu, P. Farrell, and C. Boyd, “Unified code,” in *Proc. Int. Conf. Cryptography Coding*, 1999, Lecture Notes in Computer Science, vol. 1746, pp. 84–93.
- [5] M. Grangotto, A. Grosso, and E. Magli, “Selective encryption of JPEG2000 images by means of randomized arithmetic coding,” in *Proc. IEEE 6th Workshop Multimedia Signal Processing*, Siena, Italy, Sep. 2004, pp. 347–350.
- [6] M. Grangotto, E. Magli, and G. Olmo, “Multimedia securization by means of randomized arithmetic coding,” *IEEE Trans. Multimedia*, submitted for publication.
- [7] T. Cover and J. Thomas, *Elements of Information Theory*. New York: Wiley, 1991.