A formula based approach to Arithmetic Coding

Arundale Ramanathan

Siara Logics (cc)
arun@siara.cc

Abstract

The Arithmetic Coding process involves re-calculation of intervals for each symbol that need to be encoded. This article discovers a formula based approach for calculating compressed codes and provides proof for deriving the formula from the usual approach. A spreadsheet is also provided for verification of the approach. Consequently, the similarities between Arithmetic Coding and Huffman coding are also visually illustrated.

This article presents a formula based approach to Arithmetic Coding. It also explains the mathematical foundation of Arithmetic Coding from a radically different perspective.

This article discovers a formula based approach for calculating compressed codes and provides proof for deriving the formula from the usual approach.

Using a spreadsheet, the new approach is demonstrated by compressing and decompressing a simple string ("Hello World"). Compression using conventional approach is also demonstrated in the same spreadsheet. It can be seen that the same compressed value is obtained using both the methods.

Simply put, the spreadsheet compresses the string "Hello World" having length 11 letters to a 4 byte value 2166290293 (31.298 bits to be more exact).

Radix (logarithmic) is taken as 2 in this article, but any radix can be used.

I Known facts

The basic principles of Arithmetic Coding are explained well in [2].

Referring to this article [3], for a given input string A, with N symbols (letters) and n unique symbols,

- if each unique symbol is represented as a_i, i being the index of symbol after sorting by descending order of weights,
- if each symbol appears k_i times,
- if weights (or probability) of each symbol is given by $w_i = k_i$ / N,

we know:

- the length (in bits) of optimal possible code for symbol a_i is $-log_2(w_i)$ bits, referred as $h(a_i)$.
- the total compressed length L will be $\sum_{i=1}^{n} -k_i log_2(w_i)$ or $\sum_{i=1}^{n} k_i h(a_i)$ bits.

II This work

The formula for $h(a_i)$ has been known for several decades [1]. If this indicates the length of the compressed code (in bits), what is the *value* contained in that length?

Surely it cannot be all 0s or all 1s, in which case the compressed value will simply be 0 or 1. Also, it cannot also take any arbitrary value, as there could be more than one symbol having the same compressed length. So the value is distinct and specific for each symbol. Let us call this *value* as $v(a_i)$.

If the formula for this value could be discovered, it would be simply a matter of concatenating *lengths* of *values* to obtain the code, without having to recalculate the intervals for each symbol, as required in the common approach.

III Formulae

$$v(a_i) = (\sum_{j=1}^{i-1} k_j)/k_i$$

$$compressed\_value = \sum_{j=1}^{N} (v(A[i])/2^{\sum_{j=1}^{i} h(A[j])})$$

IV Proof (derivation)

We derive the formula from the common approach, which slices the interval according to the weights w_i. So for coding any symbol a_i, the following value is used:

$$code\_value = \sum_{j=1}^{i-1} (w_j * interval\_length)$$

When we use interval as 0 to 1, interval_length is equal to 1, so it becomes:

$$code\_value = \sum_{j=1}^{i-1} w_j$$

However, since we have taken the interval as 0 to 1, the value is a fraction and we have to scale it to get the value we are seeking. The length of this value is $h(a_i)$ bits, so we shift it left as follows to get the desired value:

$$v(a_i) = (\sum_{j=1}^{i-1} w_j) * 2^{h(a_i)}$$

$$\Rightarrow v(a_i) = (\sum_{j=1}^{i-1} w_j) * 2^{-log_2(w_i)}$$

$$\Rightarrow v(a_i) = (\sum_{j=1}^{i-1} w_j) * 2^{log_2(1/w_i)}$$

$$\Rightarrow v(a_i) = (\sum_{j=1}^{i-1} w_j) * (1/w_i)$$

$$\Rightarrow v(a_i) = (\sum_{j=1}^{i-1} w_j)/w_i$$

$$\Rightarrow v(a_i) = (\sum_{j=1}^{i-1} (k_j/N))/(k_i/N)$$

$$v(a_i) = (\sum_{j=1}^{i-1} k_j)/k_i$$

V Application to other coding methods

The same formulae and approach are applicable to other coding methods such as Huffman coding, Shannon-Fano coding where *length* and *value* are available.

Once the codes for symbols are obtained using the respective methods, the Frequencies need to be re(verse)-calculated according to the code lengths ($k_i = 2^{-h(ai)} * N$). Then, the formulas can be applied to obtain the compressed value. This is shown in a separate sheet (Huffman_coding).

A picture for visual comparison between Arithmetic Coding and Huffman Coding is given under the example section below (Fig. 3).

VI Example

Given $A = "HelloWorld"$, then

- $N = 11$, $n = 8$,
- a_1 = 'l', a_2 = 'o', a_3 = 'H', a_4 = 'e', a_5 = ' ', a_6 = 'W', a_7 = 'r', a_8 = 'd', and
- $k_1 = 3$, $k_2 = 2$, k_3 to $k_8 = 1$
- w_1 = 0.2727 (3/11), w_2 = 0.1818 (2/11), w_3 to w_8 = 0.0909 (1/11)

then

- $h(a_1) = 1.8745$, $h(a_2) = 2.4594$, $h(a_3)$ to $h(a_8) = 3.4594$, and
- $L = 31.2989$

which means after compression, 11 bytes will become 31.2989 bits (around 4 bytes). By applying the formulas, we get:

- $v(a_1) = 0$, $v(a_2) = 1.5$, $v(a_3) = 5$, $v(a_4) = 6$, $v(a_5) = 7$, $v(a_6) = 8$, $v(a_7) = 9$, $v(a_8) = 10$
- $compressed\_value$ = 2166290392.64712 (0.50437878646122 unscaled)

The detailed calculations can be seen in the spreadsheet (Fig. 2).

The following picture visually shows placement of letters in compressed value for both Arithmetic and Huffman coding:

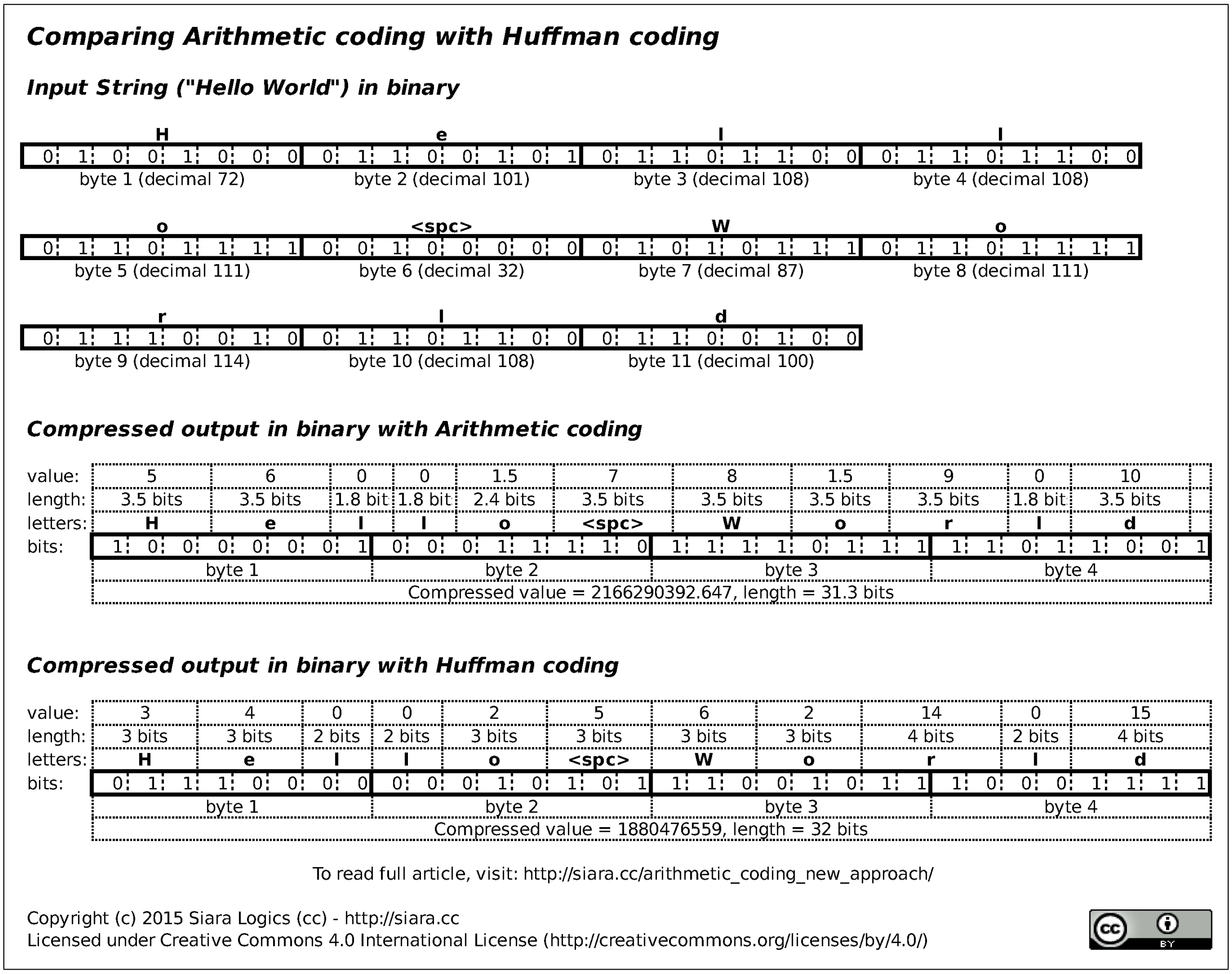

Figure 1: Visual indication of symbol positions in compressed value

The values are the same as those shown in the example spreadsheets. A screenshot of the spreadsheets are also given in the Appendix.

VII Conclusion

The current work simplifies the encoding process and explains Arithmetic coding in simpler terms. However, for practical implementations, the following points need to be considered:

- The formula based approach would heavily depend on the performance of exp2 function. It is likely to be slower than calculating the interval table for each symbol. Very little research has been done on this aspect.
- While the interval based approach does not allow parallel processing [1], the formula based approach would allow parallel processing.

Till the answers to the above points are available, this work presently serves the following purposes:

- Understand Arithmetic Coding from a different perspective
- Visualize positions of compressed symbols
- Visually compare Arithmetic coding and Huffman coding

- Precursor for further research on entropy coding

References

[1] Paul G. Howard and Jeffrey Scott Vitter, *Practical Implementations of Arithmetic Coding*, Brown University, April 1992.

[2] Wikipedia, *Arithmetic Coding*, https://en.wikipedia.org/wiki/Arithmetic_coding, September 2015.

[3] Wikipedia, *Huffman Coding*, https://en.wikipedia.org/wiki/Huffman_coding, August 2015.

VIII Appendix

Formula based approach to Arithmetic Coding

Frequency Table:

Unique Letter [ai]	Frequency (Letter count) [ki]	Weights (probability) [wi = ki / N]	Bit length [h(ai) = -log(wi)]	Total bit length (Letter Count x Bit length)	Value [v(ai)]	Lower limit (of range)	Repeat Letter
l	3	0.272727272727	1.87446911791614	5.62340735374842	0	0	l
o	2	0.181818181818	2.4594316186373	4.91886323727459	1.5	1171354717	o
H	1	0.090909090909	3.4594316186373	3.4594316186373	5	1952257862	H
e	1	0.090909090909	3.4594316186373	3.4594316186373	6	2342709434	e
	1	0.090909090909	3.4594316186373	3.4594316186373	7	2733161007	
W	1	0.090909090909	3.4594316186373	3.4594316186373	8	3123612579	W
r	1	0.090909090909	3.4594316186373	3.4594316186373	9	3514064151	r
d	1	0.090909090909	3.4594316186373	3.4594316186373	10	3904515724	d
Count [N]:	11	**Compressed length [L] (in bits)**		31.2988603028468			

To compress [A]:	Hello World
Logarithm radix:	2
No of bits:	32
Total Value:	4294967296

Compression:

Letter	Value	Bit length [h(ai)]	Bits Remaining	Compressed value
	(from frequency table)		32	
H	5	3.4594316186373	28.5405683813627	1952257861.81818
e	6	3.4594316186373	25.0811367627254	2165231446.7438
l	0	1.87446911791614	23.2066676448093	2165231446.7438
l	0	1.87446911791614	21.3321985268931	2165231446.7438
o	1.5	2.4594316186373	18.8727669082558	2165951492.6733
	7	3.4594316186373	15.4133352896185	2166256966.704
W	8	3.4594316186373	11.9539036709812	2166288704.26563
o	1.5	2.4594316186373	9.49447205234394	2166289786.22796
r	9	3.4594316186373	6.03504043370664	2166290376.38923
l	0	1.87446911791614	4.1605713157905	2166290376.38923
d	10	3.4594316186373	0.7011396971532	**2166290392.64712**

<-- Final Compressed Value

Letters are looked up -->
using compressed value -->

Usually a terminator is required when implementing for real life compression. But it is not shown in this example

Decompression:

Letter	Bits	Remaining
		(from F29)
H	3.459432	**2166290393**
e	3.459432	2354357839.12
l	1.874469	128132454.301
l	1.874469	469818999.104
o	2.459432	1722669663.38
	3.459432	3032232204.6
W	3.459432	3289783178.6
o	2.459432	1827876596.61
r	3.459432	3610870337.37
l	1.874469	1064868047.07
d	3.459432	3904516172.58

Arithmetic coding by conventional method:

Total letter count: 11

Compression:

					Lookup table for Interval (recalculated after compressing each letter)								
				Letter Count -->	3	2	1	1	1	1	1	1	
				Probability -->	**0.27273**	**0.18181818**	**0.09091**	**0.09091**	**0.09091**	**0.09091**	**0.09091**	**0.090909091**	
				Letters -->	**l**	**o**	**H**	**e**		**W**	**r**	**d**	
		Interval begin	**Interval end**	**Interval length**		**l**	**o**	**H**	**e**		**W**	**r**	**d**
1	Initial interval	0	1	1	0	0.272727273	0.454545	0.545455	0.636364	0.727273	0.818182	0.90909090909	1
2	H	0.45454545454545	0.54545454545455	0.09090909090909	0.454545	0.479338843	0.495868	0.504132	0.512397	0.520661	0.528926	0.53719008264	0.47934
3	e	0.50413223140496	0.51239669421488	0.00826446280992	0.504132	0.506386176	0.507889	0.50864	0.509391	0.510143	0.510894	0.51164537941	0.50639
4	l	0.50413223140496	0.50638617580766	0.0022539444027	0.504132	0.504746944	0.505157	0.505362	0.505567	0.505771	0.505976	0.50618127177	0.50475
5	l	0.50413223140496	0.50474694351479	0.00061471210983	0.504132	0.504299880	0.504412	0.504468	0.504523	0.504579	0.504635	0.5046910606	0.5043
6	o	0.50429988016218	0.50441164600034	0.00011176583815	0.5043	0.504330362	0.504351	0.504361	0.504371	0.504381	0.504391	0.50440148547	0.50433
7		0.50437100387737	0.50438116440811	1.016053074099E−05	0.504371	0.504373775	0.504376	0.504377	0.504377	0.504378	0.504379	0.50438024072	0.50437
8	W	0.50437839335427	0.50437931703889	9.236846127969E−07	0.504378	0.504378645	0.504379	0.504379	0.504379	0.504379	0.504379	0.50437923307	0.50438
9	o	0.50437864526826	0.50437881321092	1.679426568924E−07	0.504379	0.504378691	0.504379	0.504379	0.504379	0.504379	0.504379	0.50437879794	0.50438
10	r	0.50437878267589	0.5043787979434	1.526751425285E−08	0.504379	0.504378787	0.504379	0.504379	0.504379	0.504379	0.504379	0.50437879656	0.50438
11	l	0.50437878267589	0.50437878683976	4.163867473039E−09	0.504379	0.504378784	0.504379	0.504379	0.504379	0.504379	0.504379	0.50437878646	0.50438
12	d	**0.5043787864612**											

Compressed value: **0.5043787864612**
Scale to len [L] in F14: **2166290392.6471 <-- Compare with above method (F29)**

Figure 2: Screenshot of spreadsheet that demonstrates both methods of Arithmetic Coding

Comparison with Huffman Coding

Frequency Table:

Unique Letter [ai]	Frequency (Letter count) [ki]	Weights (probability) [wi = ki / N]	Bit length [h(ai)] (copied from binary tree built elsewhere)	Total bit length (Letter Count x Bit length)	Value [v(ai)]	Lower limit (of range)	Repeat Letter	Frequency [ki] re-calculated from h(ai)	Huffman Code
l	3	0.27272727273	2	6	0	0	l	2.75	00
o	2	0.18181818182	3	6	2	1073741824	o	1.375	010
H	1	0.09090909091	3	3	3	1610612736	H	1.375	011
e	1	0.09090909091	3	3	4	2147483648	e	1.375	100
	1	0.09090909091	3	3	5	2684354560		1.375	101
W	1	0.09090909091	3	3	6	3221225472	W	1.375	110
r	1	0.09090909091	4	4	14	3758096384	r	0.6875	1110
d	1	0.09090909091	4	4	15	4026531840	d	0.6875	1111
Count [N]:	11	**Compressed len [L] (in bits):**		32				11	

Given string [A]:	Hello World
Logarithm Radix:	2
No of bits:	32
Total Value:	4294967296

Compression:

Letter	Value	Bit length [h(ai)]	Bits Remaining	Compressed value
	(from frequency table)		32	
H	3	3	29	1610612736
e	4	3	26	1879048192
l	0	2	24	1879048192
l	0	2	22	1879048192
o	2	3	19	1880096768
	5	3	16	1880424448
W	6	3	13	1880473600
o	2	3	10	1880475648
r	14	4	6	1880476544
l	0	2	4	1880476544
d	15	4	0	**1880476559**

<-- Final Compressed Value

Usually a terminator is required when implementing for real life compression. But it is not shown in this example

Letters are looked up -->
using compressed value -->

Decompression:

Letter	Bits	Remaining
		(from F29)
H	3	**1880476559**
e	3	2158910584
l	2	91415488
l	2	365661952
o	3	1462647808
	3	3111247872
W	3	3415146496
o	3	1551368192
r	4	3821010944
l	2	1006632960
d	4	4026531840

To read full article visit:
http://siara.cc/arithmetic_coding_new_approach/

Figure 3: Screenshot of spreadsheet that demonstrates compression using Huffman codes by the same formula