ELEC S347F Multimedia Technologies

Lossless Compression Algorithms

Introduction

- Lossless compression algorithms mainly used for text and data which have zero tolerance of error
  - The Shannon-Fano Algorithm
  - Huffman Coding
  - Tunstall Coding
  - Tunstall/Huffman Coding
  - Arithmetic Coding
  - Lempel-Ziv-Welch (LZW) Coding
- Also applied in various image formats
  - RLE used in the PCX format
  - Arithmetic Coding, Huffman Coding used in the JPEG format
  - LZW (and variants) Coding used in GIF, PNG, PDF formats

The Shannon-Fano Algorithm

- Precondition: The probability model for the source is known
- The Shannon-Fano Algorithm
  - 1) Construct a list $\mathcal{S}$ of all the symbols and sort them in a descending probability order (all symbols initially have empty codewords)
  - 2) Split the list $\mathcal{S}$ into two sublists $\mathcal{S}_0$ and $\mathcal{S}_1$ such that the difference of probability between the sublists are minimized
  - 3) Assign a 0 and a 1 to the suffix of codewords in the sublists respectively
  - 4) Recursively split the sublists $\mathcal{S}_0$ and $\mathcal{S}_1$ (step 2 to 4) until all sublists contain one symbol only

The Shannon-Fano Algorithm: Example

- Consider the following probability model

Symbol	A	B	C	D	E
Probability	0.19	0.12	0.38	0.16	0.15

- The Shannon-Fano Algorithm

Symbol	C	A	D	E	B
Probability	0.38	0.19	0.16	0.15	0.12

0.57 Difference = 0.14 0.43

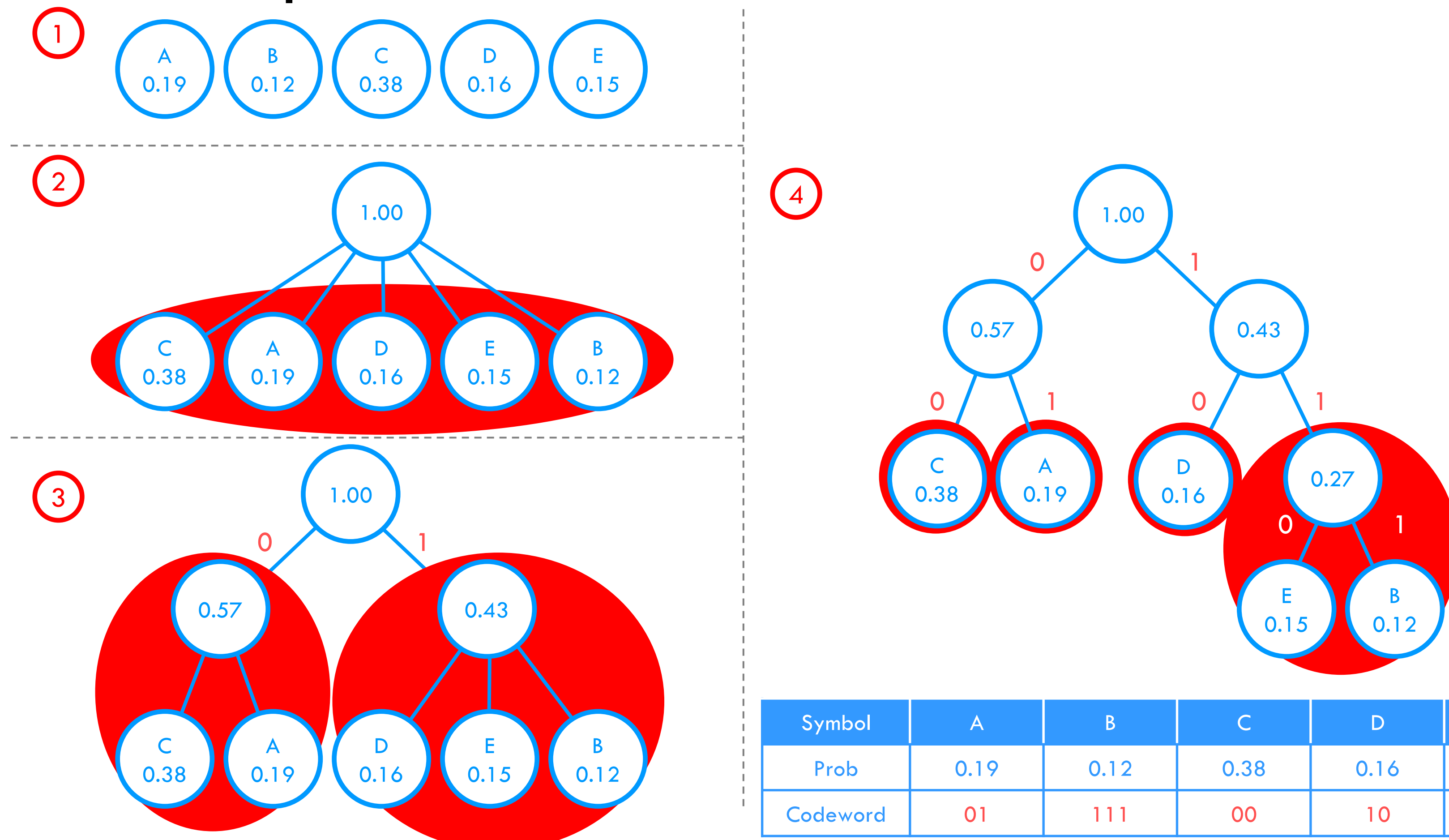
Symbol	C	A	D	E	B
Probability	0.38	0.19	0.16	0.15	0.12

0.16 Diff = 0.11 0.27

Symbol	C	A	D	E	B
Probability	0.38	0.19	0.16	0.15	0.12

The Shannon-Fano Algorithm: Example

Tree Implementation



Symbol	A	B	C	D	E
Prob	0.19	0.12	0.38	0.16	0.15
Codeword	01	111	00	10	110

The Shannon-Fano Algorithm: Example

- How to decode the encoded bit sequence?
  - Follow the bit pattern and traverse the branches (where 0 corresponds to a left branch, 1 corresponds to a right branch) from the root of the tree until meeting a leaf node, then restart from the root again
- Try to decode the string 01101011100 for the previous example
  - Is it uniquely decodable? Is it instantaneous?

How Good Shannon-Fano Coding is?

- Code Rate (r)
  - Defined as the average length per symbol
  - = average codeword length / average symbol length
  - The smaller the value, the more efficient the code
- For the previous example,
  - $r = (0.19 \times 2 + 0.12 \times 3 + 0.38 \times 2 + 0.16 \times 2 + 0.15 \times 3)/1$
  - = 2.27 bits/symbol
- How good Shannon-Fano Coding is?
  - Need to compare with entropy

Code Efficiency

- Code Efficiency (E)
  - Defined as the ratio between the entropy and the rate
  - $E = H / r$
  - The closer the value to 1, the more efficient the code
- For the previous example,
  - $H = -(0.19)\log_2(0.19) - (0.12)\log_2(0.12) - (0.38)\log_2(0.38) - (0.16)\log_2(0.16) -(0.15)\log_2(0.15)$
    - $= 2.186$ bits/symbol
  - Code Efficiency $E = H / r = 2.186 / 2.27 = 0.963$

Code Redundancy

- Code Redundancy (R)
  - Another measure of the efficiency of a code
  - Defined as the difference between the code rate and the entropy ($R = r - H$)
  - The smaller the value, the more efficient the code
- For the previous example,
  - Code Redundancy $R = r - H = 2.27 - 2.186 = 0.084$ bits/symbol
  - $0.084 > 0$ (means there are room for improvement)

Huffman Coding

Huffman Coding

- Precondition: The probability model for the source is known
- Huffman Coding Algorithm
  - 1) Construct a list $\mathcal{S}$ of all the symbols and sort them in a descending probability order (all symbols initially have empty codewords)
  - 2) Combine the two symbols with the lowest probability from $\mathcal{S}$
  - 3) Assign a 0 and a 1 to the suffix of the codewords of the two symbols respectively
  - 4) Reorder the list $\mathcal{S}$ with the aggregated probability
  - 5) Repeat step 2 until there is only one symbol left in $\mathcal{S}$

Huffman Coding: Example

- Consider the following probability model

Symbol	A	B	C	D	E
Probability	0.19	0.12	0.38	0.16	0.15

- Huffman Coding Algorithm

Symbol	C	A	D	E	B
Probability	0.38	0.19	0.16	0.15	0.12

Symbol	C	{E, B}	A	D	
Probability	0.38	0.27	0.19	0.16	

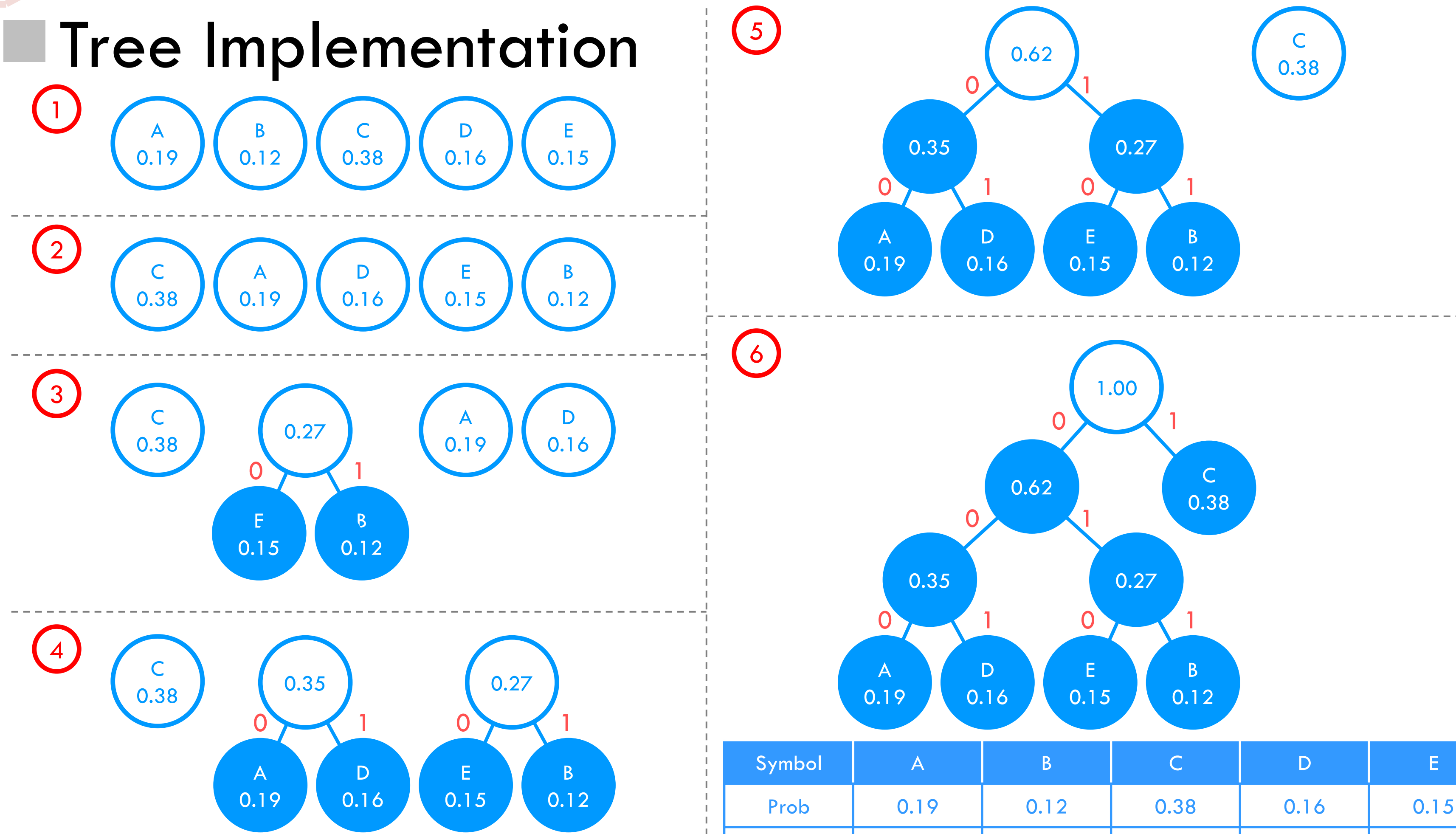
Symbol	C	{A, D}	{E, B}		
Probability	0.38	0.35	0.27		

Symbol	{{A,D}, {E,B}}	C			
Probability	0.62	0.38			

Symbol	{{{A,D},{E,B}},C}				
Probability	1				

Huffman Coding: Example

- Tree Implementation



Symbol	A	B	C	D	E
Prob	0.19	0.12	0.38	0.16	0.15
Codeword	000	011	1	001	010

Huffman Coding: Example

- How to decode the encoded bit sequence?
  - Same procedure as the Shannon-Fano algorithm
- Try decoding the string 01101011001 for the previous example
  - Is it uniquely decodable? Is it instantaneous?

How Good Huffman Coding is?

- For the previous example,
  - $r = 0.19 \times 3 + 0.12 \times 3 + 0.38 \times 1 + 0.16 \times 3 + 0.15 \times 3$
  - $= 2.24$ bits/symbol
- How good Huffman Coding is?
  - Code Efficiency $E = H / r = 2.186 / 2.24 = 0.976$
  - Code Redundancy $R = r - H = 2.24 - 2.186 = 0.054$ bits/symbol
  - Huffman code performs better than the Shannon-Fano algorithm <u>in this example</u>

Huffman Coding Algorithm

- For another example
  - If the probability model is of symbols {A, B, C, D} = {0.4, 0.2, 0.2, 0.2}
  - 3 symbols are of the same smallest probability
  - The first iteration may combine symbol {B, C}, {C, D}, {B, D} (and 3 more in reverse order)
  - The generated codes will be different (the only difference)
  - However, they are still prefix codes and uniquely decodable
  - The redundancies and efficiencies are the same

Huffman vs. Shannon-Fano

- Similarities
  - Both are prefix codes
    - Both are uniquely decodable and instantaneous
  - Both are variable-length codes
    - Symbols that occur more frequently have shorter codewords than symbols that occur less frequently
  - Both require knowledge of the probability model of the source data
    - The bit assignments are based on the probability of the symbols
  - Both require the coding table for decoding
    - An overhead that need to send before the data
    - Negligible only if the source data is big enough

Huffman vs. Shannon-Fano

- Differences
  - The Shannon-Fano algorithm uses top-down approach while Huffman Coding uses bottom-up approach
  - The Shannon-Fano algorithm does not guarantee to generate the optimal tree
    - In contrast, the Huffman algorithm always
    - So Huffman coding is able to achieve higher code efficiency

Knowledge of Source Data

- Knowledge of the probability model of the source data
  - The first pass (modeling): collect the statistics (the occurrence probabilities of the symbols)
  - The second pass (coding): encode the source based on the collected statistics
- For some applications (e.g. compressing an article)
  - Modeling may use global statistics (e.g. occurrence of letters based on English text)
  - Or use local statistics (e.g. occurrence of letters based on the processing article)
  - What are the pros and cons?

No Knowledge of Source Data

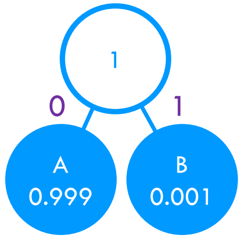
- What if the collection of statistics is not possible?
  - e.g. live application
- Two possible directions
  - Using global statistics (provided that is available)
    - In general, the code efficiency is not high
  - Building the code table dynamically and adaptively
    - Representatives: dynamic Huffman coding, adaptive arithmetic coding, Lempel-Ziv-Welch (LZW) coding

Limitation of Huffman Coding

- Consider the following probability model

Symbol	A	B
Probability	0.999	0.001

- Huffman Coding Algorithm



  - Entropy H = 0.0114 bits/symbol
  - Code Rate r = 0.999 x 1 + 0.001 x 1 = 1 bit/sym
  - r >> H

Why Does Huffman Coding Fail?

- Huffman codes use an integer number of bits for each symbol
  - If there are 999 symbol 'A's and 1 symbol 'B'
  - Using Huffman coding requires 1000 bits for the whole sequence
- The ideal case of the previous example should use less than 1 bit to represent symbol 'A'
  - i(A): $-\log_2(0.999) = 0.00144$
  - i(B): $-\log_2(0.001) = 9.966$
  - Total length = $0.00144 \times 999 + 9.966 \times 1 = 11.4$ bits
  - $<<$ 1000 bits

Solution

- Use real number of bits for each symbol
  - Not possible?
  - What if assigning integer number of bits for a group of symbols?
  - e.g. y bits for a group of x symbols
  - Each symbol uses y/x bits (which approximate to the value of self-information)
  - Representatives: Tunstall Codes, Arithmetic Codes

Tunstall Coding

Tunstall Codes

- Variable-length codes means that the length of the codewords are vary in number of bits
  - Usually fewer bits are assigned for symbols that occur more frequently
  - But each codeword corresponds to a symbol
- Tunstall codes are fixed-length code
  - However, each codeword represents a different number of symbols
  - Called variable-length to fixed-length code
  - Maximize the (average) number of symbols represented by each codeword (so as to minimize the overall length of encoded string)

Tunstall Coding Algorithm

- Algorithm for generating k-bit Tunstall Codes
  - 1) Construct a list $\mathcal{S}$ of all the symbols
  - 2) Remove the symbol φ that has the highest probability from $\mathcal{S}$
  - 3) Concatenate φ with every source symbols (including φ itself) and add them to the list $\mathcal{S}$
  - 4) Repeat step 2 if the number of symbols in $\mathcal{S}$ is less than or equal to 2^k
  - Then assign the bits to the symbols

Tunstall Coding Algorithm: Example

- Design a 3-bit Tunstall code for the below 3 symbols

Symbol	Probability
A	0.6
B	0.3
C	0.1

  - For generating k-bit Tunstall codes for N symbols
    - Require M iterations, where $N + M(N-1) \leq 2^k$

Symbol	Probability
~~A~~	~~0.6~~
B	0.3
C	0.1
AA	0.36
AB	0.18
AC	0.06

Symbol	Probability
~~A~~	~~0.6~~
B	0.3
C	0.1
~~AA~~	~~0.36~~
AB	0.18
AC	0.06
AAA	0.216
AAB	0.108
AAC	0.036

Symbol	Codeword
B	000
C	001
AB	010
AC	011
AAA	100
AAB	101
AAC	110

Code Efficiency and Redundancy

- For the previous example,
  - $H = -(0.6)\log_2(0.6) - (0.3)\log_2(0.3) - (0.1)\log_2(0.1) = 1.30$ bits/symbol
  - r = avg. codeword length / avg. symbol length
    - $= 3/(1\text{x}0.3 + 1\text{x}0.1 + 2\text{x}0.18 + 2\text{x}0.06 + 3\text{x}0.216 + 3\text{x}0.108 + 3\text{x}0.036)$
    - $= 3/1.96$
    - $= 1.53$ bits/symbol
  - Code Efficiency = H / r = 0.85
  - Code Redundancy = r – H = 0.23 bits/symbol

Tunstall Coding Remarks

- Tunstall codes are uniquely decodable
- However, it is not uniquely encodable
- For the previous example, input symbols “A” and “AA” have no corresponding codewords
  - To ensure the unique encodability, no input code should be a prefix of codeword in Tunstall codes
  - How about if such prefixes really exist?
  - Assign the unused codes (e.g. “111” in the previous example) for the prefixes

Tunstall/Huffman Coding

- Construct the probability model of symbols using Tunstall Coding
- Then encode the symbols using Huffman Coding
- Variable-length to variable-length code
- For the previous example:

Symbol	Probability
~~A~~	~~0.6~~
B	0.3
C	0.1
AA	0.36
AB	0.18
AC	0.06

Symbol	Probability
~~A~~	~~0.6~~
B	0.3
C	0.1
~~AA~~	~~0.36~~
AB	0.18
AC	0.06
AAA	0.216
AAB	0.108
AAC	0.036

Symbol	Codeword
B	00
C	110
AB	010
AC	1110
AAA	10
AAB	011
AAC	1111

Code Efficiency and Redundancy

- For the previous example,
  - r = avg. codeword length / avg. symbol length
    - = (2x0.3 + 3x0.1 + 3x0.18 + 4x0.06 + 2x0.216 + 3x0.108 + 4x0.036) / (1x0.3 + 1x0.1 + 2x0.18 + 2x0.06 + 3x0.216 + 3x0.108 + 3x0.036)
    - = 2.58/1.96
    - = 1.32 bits/symbol
  - Code Efficiency = H / r = 0.98
  - Code Redundancy = r – H = 0.02 bits/symbol
  - More efficient than using Tunstall coding alone!

Compare with Huffman Coding

- What if using Huffman coding only,

Symbol	Probability	Codeword
A	0.6	0
B	0.3	10
C	0.1	11

  - r = avg. codeword length / avg. symbol length
    - = (1x0.6 + 2x0.3 + 2x0.1) / 1
    - = 1.4 bits/symbol
  - Code Efficiency = H / r = 0.93
  - Code Redundancy = r – H = 0.1 bits/symbol
  - Tunstall/Huffman coding is better than using Tunstall and Huffman coding alone!

Tunstall/Huffman Coding: Conclusion

- Having a good model for the data can be useful in estimating the entropy of the source
- The better the modeling, the higher efficiency of the compression algorithm
- Tunstall coding provides a better modeling of the source data, so as to allow better encoding rate for Huffman coding
  - Maximize the number of symbols per codeword and minimize the number of bits per codeword
- Any trade off?

Arithmetic Coding

Arithmetic Coding

- Idea: encode the entire message into a single real number in range [0, 1)
- Encode algorithm
  - Initialize the range as [0, 1)
  - For each input symbol K, narrow down the range based on its probability model
  - Finally output a real number that falls in the range
- Decode algorithm
  - Initialize the range as [0, 1)
  - Based on the encoded real number, find the range of number within which the code number lies, and update the range based on the decoded symbol
  - Repeat until all symbols have been decoded

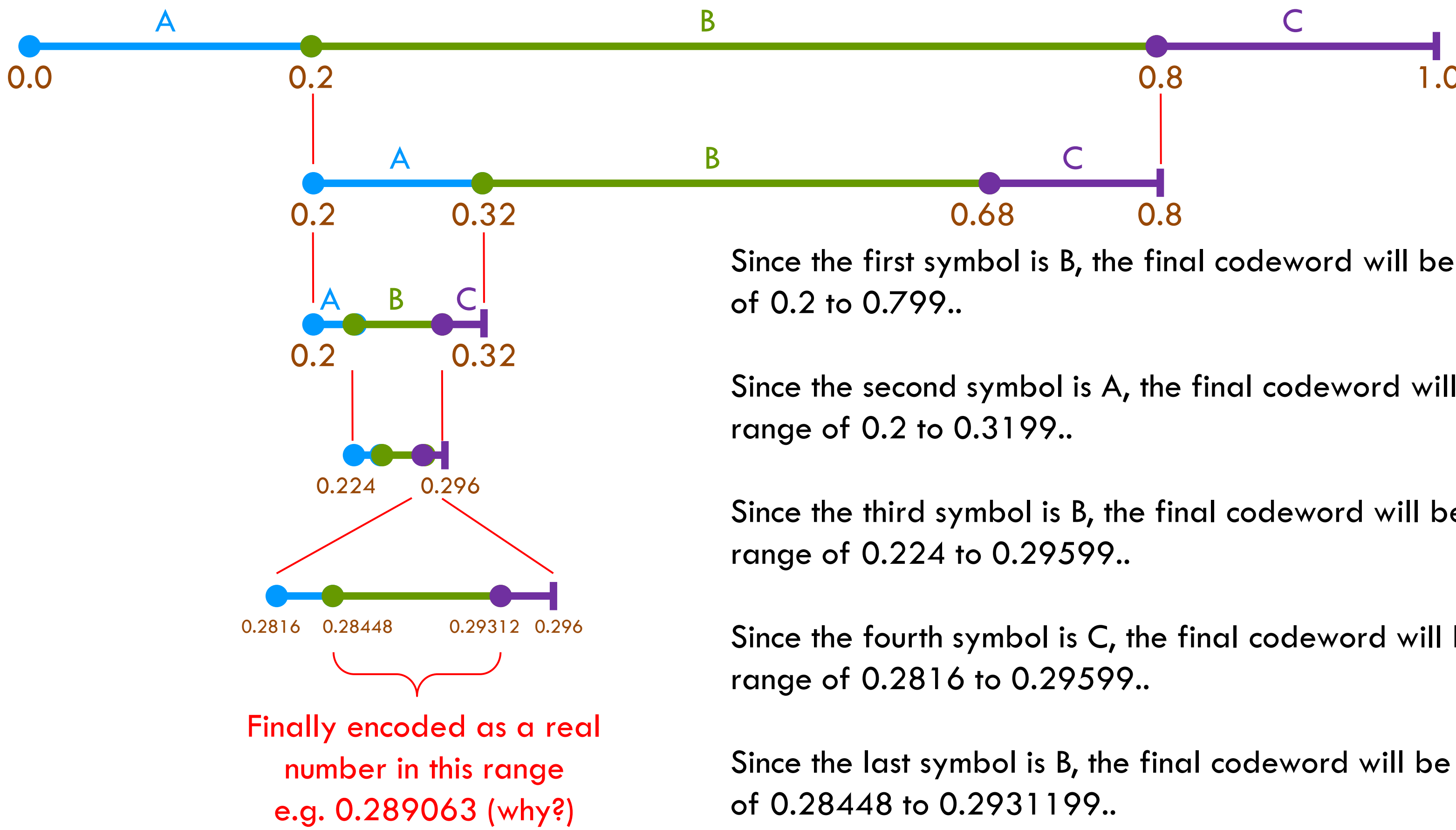
Arithmetic Encoding: Example

- If the input sequence = {B, A, B, C, B}
  - Probability model = {A: 0.2, B: 0.6, C: 0.2}
- Encoding
  - Assign each symbol to the probability range
    - A: [0.0, 0.2), B: [0.2, 0.8), C: [0.8, 1.0)

  - Since the first symbol is B, the final codeword will be in the range of 0.2 to 0.79999..

Arithmetic Encoding: Example

- Input sequence = {B, A, B, C, B}



Since the first symbol is B, the final codeword will be in the range of 0.2 to 0.799..

Since the second symbol is A, the final codeword will be in the range of 0.2 to 0.3199..

Since the third symbol is B, the final codeword will be in the range of 0.224 to 0.29599..

Since the fourth symbol is C, the final codeword will be in the range of 0.2816 to 0.29599..

Since the last symbol is B, the final codeword will be in the range of 0.28448 to 0.2931199..

Efficiency of Arithmetic Coding

- For the previous example, the sequence could be encoded as $0.289063_{10} = 0.0100101_2$
  - $=2^{-2} + 2^{-5} + 2^{-7} = 0.25 + 0.03125 + 0.007813$
  - Require 7 bits for the sequence only
  - r = 7/5 = 1.4 bits/sym
- Comparison
  - $H = -(0.2)\log_2(0.2) - (0.6)\log_2(0.6) - (0.2)\log_2(0.2) =$ 1.37 bits/sym
  - Total length = 1.37 x 5 = 6.85 bits
  - E = H/r = 0.979
  - R = r – H = 0.0214 bits/symbol

Note: coding requires integer no. of bits for the whole sequence, in which at least 7 bits in this example.
It is already the best that arithmetic coding can do

Arithmetic Decoding: Example

- Given the probability model of a symbol set
  - {A: 0.2, B: 0.6, C: 0.2}
  - If the encoded number is 0.289063, what are the original source symbols?
- Decoding processing

Symbol	Range
A	[0.0, 0.2)
B	[0.2, 0.8)
C	[0.8, 1.0)

Since 0.289063 falls in between [0.2, 0.8), the first symbol is decoded as ‘B’

Symbol	Range
BA	[0.20, 0.32)
BB	[0.32, 0.68)
BC	[0.68, 0.80)

Since 0.289063 falls in between [0.20, 0.32), the second symbol is decoded as ‘A’

Arithmetic Decoding: Example

- Decoding processing

Symbol	Range
BAA	[0.200, 0.224)
BAB	[0.224, 0.296)
BAC	[0.296, 0.320)

Since 0.289063 falls in between [0.224, 0.296), the third symbol is decoded as 'B'

Symbol	Range
BABA	[0.2240, 0.2384)
BABB	[0.2384, 0.2816)
BABC	[0.2816, 0.2960)

Since 0.289063 falls in between [0.2816, 0.2960), the fourth symbol is decoded as 'C'

Symbol	Range
BABCA	[0.28160, 0.28448)
BABCB	[0.28448, 0.29312)
BABCC	[0.29312, 0.29600)

Since 0.289063 falls in between [0.28448, 0.29312), the fifth symbol is decoded as 'B'

- The decoded sequence = {B, A, B, C, B}

Arithmetic vs. Huffman Coding

- Huffman coding uses integer no. of bits to represent symbols
  - In contrast, arithmetic coding uses real range to represent symbols
  - The no. of bits is determined by the size of the interval ($-\log_2 p$ bits to represent interval of size p)
  - Asymptotically arithmetic coding approaches ideal entropy
  - i.e. able to achieve better compression rate than Huffman coding
- However, arithmetic coding is much more complex
  - Computation can be memory and FPU intensive
  - Need to renormalize the floating point numbers dynamically
  - Otherwise the resolution of number would be limited by FPU precision

Lempel-Ziv-Welch (LZW) Coding

Lempel-Ziv-Welch (LZW) Encoding

- initialize the dictionary (i.e. code table) with all basic symbols
- W = “” (null string)
- while (read an input symbol K)
  - if WK is in dictionary
    - W = WK
  - else
    - append WK to the dictionary
    - output the codeword for W
    - W = K
- output the codeword for W

LZW Encoding: Example

- Consider a simple example with 3 possible basic symbols
  - Symbol Set: {a, b, c}
  - Input Sequence: {a, b, a, b, a, c, a, b, a}
- Encoding
  - Initialize the dictionary (code table) with all possible basic symbols

Symbol	Codeword
a	0
b	1
c	2

LZW Encoding: Example

- Input sequence: {a, b, a, b, a, c, a, b, a}

W	K	WK	Output
“” (null)	a	a	-

Accumulated output = “”

Symbol	Codeword
a	0
b	1
c	2

W	K	WK	Output
a	b	ab	0

Accumulated output = 0

Symbol	Codeword
a	0
b	1
c	2
ab	3

W	K	WK	Output
b	a	ba	1

Accumulated output = 0, 1

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4

LZW Encoding: Example

- Input sequence: {a, b, a, b, a, c, a, b, a}

W	K	WK	Output
a	b	ab	-

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4

Accumulated output = 0, 1

W	K	WK	Output
ab	a	aba	3

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4
aba	5

Accumulated output = 0, 1, 3

LZW Encoding: Example

- Input sequence: {a, b, a, b, a, c, a, b, a}

W	K	WK	Output
a	c	ac	0

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4
aba	5
ac	6

Accumulated output = 0, 1, 3, 0

W	K	WK	Output
c	a	ca	2

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4
aba	5
ac	6
ca	7

Accumulated output = 0, 1, 3, 0, 2

LZW Encoding: Example

- Input sequence: {a, b, a, b, a, c, a, b, a}

W	K	WK	Output
a	b	ab	-

Accumulated output = 0, 1, 3, 0, 2

Symbol	Codeword
a	0
b	1
c	2
ab	3
ba	4
aba	5
ac	6
ca	7

W	K	WK	Output
ab	a	aba	-

Accumulated output = 0, 1, 3, 0, 2

W	K	WK	Output
aba	-	-	5

Accumulated output = 0, 1, 3, 0, 2, 5

Performance

- In the previous example
  - The final code table has 8 entries
    - 3 bits per codeword
  - The encoded sequence
    - = {0, 1, 3, 0, 2, 5}
  - Encoded bit sequence
    - = {000 001 010 000 010 101}
  - Total encoded length = 6 x 3 = 18 bits
  - The input = 9 x 2 =18 bits (No compression?)

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)
ba	4 (100)
aba	5 (101)
ac	6 (110)
ca	7 (111)

LZW Decoding

- initialize the dictionary (i.e. code table) with all basic symbols
- W = “” (null string)
- while (read a codeword K)
  - if K exists in the dictionary
    - output the symbol of K
  - else
    - output W + first symbol W
  - append “W + first symbol of K” to the dictionary if it does not exist
  - W = the symbol of K

LZW Decoding: Example

- Given the initial dictionary with 3 entries
- And the encoded sequence
  - = {0, 1, 3, 0, 2, 5}
- How to reconstruct the original source?

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)

W	K	W+sym(K[0])	Output
-	0	a	a

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)

Accumulated output = a

W	K	W+sym(K[0])	Output
a	1	ab	b

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)

Accumulated output = ab

LZW Decoding: Example

- Encoded sequence = {0, 1, 3, 0, 2, 5}

W	K	W+sym(K[0])	Output
b	3	ba	ab

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)
ba	4 (101)

Accumulated output = abab

W	K	W+sym(K[0])	Output
ab	0	aba	a

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)
ba	4 (100)
aba	5 (101)

Accumulated output = ababa

LZW Decoding: Example

W	K	W+sym(K[0])	Output
a	2	ac	c

Accumulated output = ababac

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)
ba	4 (100)
aba	5 (101)
ac	6 (110)

W	K	W+sym(K[0])	Output
c	5	ca	aba

Accumulated output = ababacaba (decoded data)

The final decoded sequence and code table are the same as the source!

Symbol	Codeword
a	0 (000)
b	1 (001)
c	2 (010)
ab	3 (011)
ba	4 (100)
aba	5 (101)
ac	6 (110)
ca	7 (111)

LZW Coding Analysis

- Effective for source data with large symbol set and repeated pattern
- Require good use of data structures for maintaining the dictionary
  - Otherwise will waste time on matching the coded symbols
- Original LZW used bounded sized dictionary
  - 12-bit fixed-length codewords (so max. 4096 entries)
  - First 256 entries are ASCII codes
  - The remaining are built on-the-fly