EE669 Homework #1

Yifan Wang
wang608@usc.edu

September 14, 2019

1 Problem 1: Written Questions

1.1 Huffman Coding

(1)

$$H = E[-log(p_i)] = -\sum p_i log_2(p_i)$$
$$= -(\frac{400}{1000}log_2(\frac{400}{1000}) + \frac{200}{1000}log_2(\frac{200}{1000}) + \frac{200}{1000}log_2(\frac{200}{1000}) + \frac{100}{1000}log_2(\frac{100}{1000}) + \frac{100}{1000}log_2(\frac{100}{1000}))$$
$$= 2.1219$$

(2)

$$Avg - length = \lceil log_2(5) \rceil = 3$$

(3)

It it not unique, 2 trees are shown in $Figure1$.

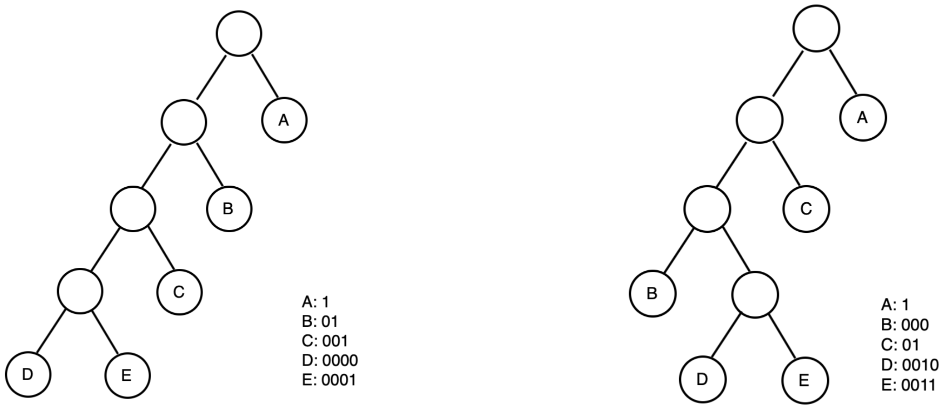


Figure 1: Huffman Coding Trees

(4)

1. D, E $---> 0, 1$. total freq: 200. State: A: 400, B: 200, C: 200, (D, E): 200.
2. (D, E), C $---> 0, 1$, total freq: 400, State: A: 400, ((D, E), C): 400, B: 200.
3. ((D, E), C), B $---> 0, 1$, total freq: 600, State: A: 400, ((C, (D, E)), B): 600.
4. ((D, E), C), B), 1 $---> 0, 1$. Done

Res A: $(1)_2$, B: $(01)_2$, C: $(001)_2$, D: $(0000)_2$, E: $(0001)_2$

$$Avg - length = 1 * 0.4 + 2 * 0.2 + 3 * 0.2 + 4 * 0.1 + 4 * 0.1 = 2.2$$

$$File - size = 400 * 1 + 200 * 2 + 200 * 3 + 100 * 4 + 100 * 4 = 2200(bit) = 275(byte)$$

The compression result would have the same file size for different tree.

(5)

Coding redundancy is:

$$R = 2.2 - 2.1219 = 0.0781$$

(6)

Encoding result for $AEDCAEBBCADBCABCAA$ using the left tree in $Figure1$ is:

$$1000100000011000101010011000001001101001111$$

1.2 Lempel-Ziv-Welch Coding

(1)

Input is $ababbaabbcd$.

Dictionary	Input	Buffer	Output Phrase	Output Index
1: a				
2: b				
3: c				
4: d				
	a	a		
	b	ab	a	1
5: ab	a	ba	b	2
6: ba	b	ab		
	b	abb	ab	5
7: abb	a	ba		
	a	baa	ba	6
8: baa	b	ab		
	b	abb		
	c	abbc	abb	7
9: abbc	d	cd	c	3
10: cd	EOF	d	d	4

Final coding result is $1, 2, 5, 6, 7, 3, 4$

(2)

Decoding: $1, 5, 2, 3, 1, 2, 9, 10, 8, 4$.

Dictionary	Input	Buffer	Output Phrase	Left
1: a				
2: b				
3: c				
4: d				
	1	a	a	
	5	aa		aa
5: aa	2	aab	aa	b
6: aab	3	bc	b	c
7: bc	1	ca	c	a
8: ca	2	ab	a	b
9: ab	9	bab	b	ab
10: ba	10	abba	ab	ba
11: abb	8	baca	ba	ca
12: bac	4	cad	ca	d
13: cad		d	d	

The decoding result is:

$$aaabcababbacad$$

1.3 Arithmetic Coding

(1)

Generally arithmetic coding is more efficient than Huffman coding. Since for Huffman coding, no matter how high the probability of one symbol is, it uses at least one bit to encode, and need integer number of bit for each other symbol which would be nice when the probability is approaching $2^{-n}, (n \in Z^{+})$. On the other hand, arithmetic coding can approaching the entropy by fraction bit and reduce more redundancy, and resulting a smaller compressed file.

(2)

$$A : [0, 0.25), B : [0.25, 0.4), C : [0.4, 0.6), D : [0.6, 1)$$

Step	Input	Low	High
0		0	1
1	C	0+(1-0)*0.4 = 0.4	0+(1-0)*0.6=0.6
2	A	0.4+(0.6-0.4)*0=0.4	0.4+(0.6-0.4)*0.25=0.45
3	B	0.4+(0.45-0.4)*0.25=0.4125	0.4+(0.45-0.4)*0.4=0.42
4	D	0.4125+(0.42-0.4125)*0.6=0.417	0.4125+(0.42-0.4125)*1=0.42
5	D	0.417+(0.42-0.417)*0.6=0.4188	0.417+(0.42-0.417)*1=0.42

The result is in range $[0.4188, 0.42)$, One binary coding result for $CABDD$ can be:

$$\{0.419875\}_{10}$$

,

$$\{0.0110101101111100111011011001000101101000011100101011 01\}_2$$

(3)

Step	Input	Output
1	0.650927	D
2	$\frac{0.650927-0.6}{1-0.6} = 0.1273175$	A
3	$\frac{0.1273175-0}{0.25-0} = 0.50927$	C
4	$\frac{0.50927-0.4}{0.6-0.4} = 0.54635$	C
5	$\frac{0.54635-0.4}{0.6-0.4} = 0.73175$	D
...	...	...

Result for decoding $\{0.650927\}_{10}$ is:

$$DACCD$$

2 Problem 2: Entropy Coding

2.1 Shannon-Fano Coding

(1)

It is based on the probability of a symbol in the file, using different length of code to represent different symbol. The higher probability the symbol is, the shorter code length it has. It's procedure is:

1. Give the frequency of each symbol in the file.
2. Sort the frequency.
3. Separate all the symbols (sorted by frequency in $Step2$) in to 2 group (0 and 1) where the total frequency of each group is closest.
4. Repeat $Step3$ until each group has only one symbol.
5. Concat the value of each group results the Shannon code for each symbol.

The following part is used to searching the splitting boundary. $symbol$ represents the sorting result of each symbol. $highPtr$ and $lowPtr$ represent the searching region, $highSum$ and $lowSum$ save the accumulate frequency.

```
while ( highPtr != lowPtr - 1 ) {
      if ( highSum > lowSum ) {
          lowPtr --;
          lowSum += symbol.at( lowPtr ).getFrequency();
      } else {
          highPtr ++;
          highSum += symbol.at( highPtr ).getFrequency();
      }
  }
```

(2)

Realized in $ShannonFano.hpp$.

(3)

File Name	Raw Size (byte)	Entropy	Shannon-Fano (byte)	SF code length	Compression ratios
audio.dat	65536	6.45594	53356	6.51318	0.8141
binary.dat	65536	0.183244	8192	1	0.125
image.dat	65536	7.59311	62607	7.64238	0.9553
text.dat	8700	4.435	4866	4.47368	0.5593

Table 1: Compression result for Shannon-Fanno Coding

From $Table1$ shows the compress result. It can be found that Shannon-fano encoder would lower the file size but remains a lot redundancy. This gap is extremely large for the binary data where the symbol distribution is extreme (more than 90% is 1), the coding redundancy is large. That's a result of Shannon-fano coder need at least one bit for each symbol. That would be a problem in the case like $binary.dat$ where only two symbol occurs.

For the other cases ,the redundancy is about 0.05, which comes from that the distribution of the symbols is not prefect. Since Shannon-fano would approach to max performance when the symbol distribution follows 2^n. In one word, Shannon-fano encoder is more important in theory rather than in real world user case, due to its high redundancy.

2.2 Huffman Coding with Global Statistics

(1)

Realized in $Huffman\_GS.hpp$.

(2)

File Name	Raw Size (byte)	Entropy	Huffman (byte)	Huffman code length	Compression ratios
audio.dat	65536	6.45594	53166	6.48994	0.8112
binary.dat	65536	0.183244	8192	1	0.125
image.dat	65536	7.59311	62433	7.62111	0.9527
text.dat	8700	4.435	4864	4.47184	0.5590

Table 2: Compression result for Huffman coding

Info: The dictionary for each file is shown in $Appendices$ at the end of this report.

Result for Huffman coding would reduce the file size but remains keeping some redundancy. Due to the fact that for Huffman coding needs integer bit for one singe symbol (at least one bit). It would perform bad on $binary.dat$ compare to the ideal entropy. While for the other cases, the entropy and the average code length for Huffman coding is approaching. The small redundancy comes from the difference of symbol distribution from ideal distribution.

(3)

The compression result for Shannon-Fano (based on cumulative distribution function) and Huffman coding (based on symbol frequency) is almost the same, but Huffman coding would perform better in most case. Both method needs integer bits to represent a symbol which would limits their performance. While Huffman coding would promise the symbol which has larger frequency has shorter code length. That is a reason why Huffman coding would perform slightly better than Shannon-Fano coding (can only promise larger frequency would generally have shorter code length but may not optimial).

For the *binary.dat*, since both methods needs at least one byte to represent a symbol, the compression result is the same. For the other data (all contain 1 byte symbols), if more symbols appear in the data, the more redundancy Huffman code can reduce which is caused by the fact that Huffman coding would always produce the optimal code which has shorter (or at least the same) average code length than Shannon-Fanno coding.

2.3 Huffman Coding with Locally Adaptive Statistics

(1)

Adaptive Huffman Coding would maintain a tree (dictionary) when sweep through the data file. It only needs one sweep and no longer need to transmit coding dictionary or store it in memory while decoding. Besides, for some file, it would be impossible to get the statics of the whole file.

The detail procedures are as following:

1. Setup an empty node in the tree before sweeping the data.
2. Input a symbol, if it appears for the first time, use a tree include this symbol to replace empty node. And increase the weight for the related node.
3. If this symbol exists on the tree, go to leaf which represent it.
4. Only increase its weight and check if it fits the rules (each node is listed in order of increasing weight, from left to right, from leaf to root). Which means if a node has larger weight than its parent, other node in the same/higher level as their parents or its right node brother, they should be swapped.
5. Increase its parents's weight, and check if parents meets the requirement in $Step4$.
6. Repeat $Step5$ until the parents node is root.

(2)

File Name	Entropy	Huffman[1] (byte)	Compression ratios	Huffman[2] (byte)	Compression ratios
audio.dat	6.45594	53660	0.8188	53410	0.8134
binary.dat	0.183244	8423	0.1285	8420	0.1282
image.dat	7.59311	62904	0.9598	62673	0.9545
text.dat	4.435	4947	0.5686	4877	0.5506

Table 3: Compression result for adaptive Huffman coding
[1] with default 1bit offset and dictionary stored in the file, [2] compressed data only

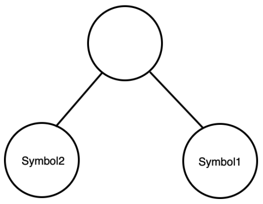
(3)

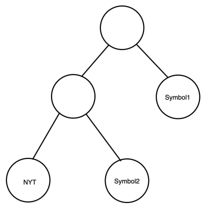
Only one search in data is a super nice advantage however it needs super long time to search and update the tree which make it even cost more time than a two search method which is a reason why it is generally not used in real world examples.

The result of adaptive Huffman is close to Huffman, but in the test cases, it perform slightly worse than Huffman and Shannon-Fanno. The reason why these cases did not perform well may lies in that the symbol changes too quick for the tree to adapt. For example, $AAAABBBCDD$ using Huffman or Shannon will reduce it to 3 bit. While using adaptive Huffman is 8 bit (one bit comes from the offset defined in the code). It seems that in our previous method, we do not store a dictionary in the code the code would just look like $1, 1, 1, 1, 01, 01, 01, 001, 0000, 0000$ for this case. While this is just to verify the data compression theory. If lost the dictionary, it would no longer be readable. No one knows which 1 means what. However in this case, it would store the ASCII code for a char when it first appears, so that it would be possible to decode the file. In one word, it also store the ASCII code for each symbol in the file, so that it would be much larger.

Even though remove the dictionary and the offset, it can be found that the result is still larger than the global Huffman coding or Shannon-Fanno coding. One reason can be the algorithm cannot adaptive to the change of symbol statics resulting longer coding length. That is a tradeoff of one sweep.

Another reason is the existence of NYT node. The gap is extremely large when it is applied on $binary.dat$ where the previous method use only one byte for each symbol. While in adaptive Huffman, due to the existence of the NYT node, some symbols would have a code length of 2 during the process which result a longer average code length and worse compression result. It is the same case for the other file. There would always a space for the NYT node even if all the symbols has been met. One subtree may looks like this,





It is obvious that with a NYT node existing in the dictionary tree would give longer code length and result a worse compression result.

Appendices

char	code	char	code	char	code	char	code	char	code
0	10010100100100	1	100101000011001	3	11111111100000010	8	11111111100000011	10	111111111000001
11	111111111000010	12	100101000011010	13	11010000100000	14	0111010110111000	15	11010000100001
16	0111010110111001	17	111111111000011	18	11010000100010	19	10010100100101	20	11111111100110
22	10010100100110	23	0111010110111010	24	100101000011011	25	11111111100111	26	10010100100111
27	11010000100011	28	11111111101110	29	11010000100100	30	1001101100101	31	11111111101111
32	10010100101000	33	1110010101000	34	11010000100101	35	1101000010101	36	1010011111100
37	0111010001000	38	10010100101001	39	011101000101	40	1101000010110	41	111111111000100
42	0111010001001	43	100101001111	44	011101011010	45	1111111110100	46	010100001010
47	1111111110101	48	1111111110110	49	101100100100	50	111001010110	51	100110110011
52	100110110000	53	101100101101	54	100101000010	55	01110100000	56	110100001100
57	10010100000	58	111111111100	59	01110100011	60	10010100010	61	11101110010
62	0111111100	63	11111111111	64	11111100100	65	11010000111	66	0101000011
67	1001101101	68	0111010111	69	0111111110	70	1011101111	71	1011001000
72	1101000001	73	1010011110	74	1110111000	75	1101000000	76	001001111
77	010100000	78	001001110	79	100001000	80	101010010	81	010101001
82	111000001	83	100110111	84	111101101	85	111001011	86	111101110
87	00100110	88	01010001	89	111111011	90	111111000	91	01111110
92	10100100	93	10100110	94	10000101	95	10011010	96	10101000
97	10110011	98	11010001	99	10111110	100	11100010	101	11100100
102	11100011	103	11111110	104	0101001	105	0111011	106	1001100
107	1000011	108	1010101	109	1011110	110	1011100	111	1110110
112	1110011	113	001100	114	011100	115	011110	116	100010
117	101000	118	101011	119	110010	120	110011	121	110101
122	111010	123	111110	124	00000	125	00001	126	00111
127	01011	128	01001	129	01100	130	01000	131	00101
132	00011	133	00010	134	111100	135	110111	136	110110
137	110000	138	101101	139	100111	140	100100	141	100011
142	100000	143	011010	144	001101	145	001000	146	1111010
147	1101001	148	1100010	149	1100011	150	1011000	151	1001011
152	0111110	153	0101011	154	0110110	155	0110111	156	0010010
157	11101111	158	11100001	159	10111111	160	10111010	161	10100101
162	10010101	163	01010101	164	111101111	165	111111110	166	111111010
167	111101100	168	111011101	169	111000000	170	101110110	171	101001110
172	101010011	173	010101000	174	100001001	175	011101010	176	1111110011
177	0111111101	178	1011101110	179	1011001010	180	1110010100	181	0111010010
182	0111010011	183	0111111111	184	11111100101	185	10110010011	186	111011100110
187	110100001101	188	111111111101	189	01010000100	190	10110010111	191	10010100011
192	111001010111	193	01110101100	194	01110100001	195	111011100111	196	010100001011
197	101100100101	198	1110010101001	199	101001111100	200	1110010101010	201	101001111101
202	1101000010111	203	1010011111101	204	1010011111110	205	100110110001	206	1001010011010
207	0111010110110	208	1110010101011	209	101100101100	210	1001010011011	211	1001010011100
212	111111111000101	213	11010000100110	214	1001010011101	215	1010011111111	216	100101000011100
217	0111010110111011	218	111111111000110	219	100101000011101	220	10010100101010	221	11010000100111
222	10010100101011	223	10010100101100	224	10010100101101	225	10010100101110	226	10010100101111
227	100101000011110	228	111111111000111	229	100101000011111	230	0111010110111100	231	100101001000000
232	111111111001000	233	1001101100100	234	10010100110000	235	100101001000001	236	10010100110001
237	111111111001001	238	0111010110111101	239	0111010110111110	240	0111010110111111	241	11010000101000
242	100101001000010	243	11010000101001	244	111111111001010	245	100101001000011	246	100101001000100
247	10010100110010	248	100101001000101	249	10010100110011	250	1001010000110000	251	100101001000110
252	1001010000110001	253	100101001000111	254	111111111001011	255	1111111110000000		

Table 4: Char Code Dictionary for $audio.dat$

char	code	char	code
0	0	255	1

Table 5: Char Code Dictionary for $binary.dat$

char	code	char	code	char	code	char	code	char	code
3	110011101101011	4	100110110011	5	10011011000	6	11011100110	7	1001101101
8	1101010010	9	010010110	10	101000111	11	111011101	12	00100000
13	10000001	14	11001000	15	11011101	16	11111110	17	0011011
18	0000001	19	1001001	20	0110000	21	1001000	22	1001111
23	1000101	24	1011110	25	1011001	26	0100010	27	0010001
28	0111111	29	0110001	30	0000111	31	0000110	32	11011110
33	11010111	34	11010101	35	11001100	36	11001001	37	01110101
38	10110001	39	11000001	40	10011010	41	10001001	42	01110111
43	01010110	44	11000000	45	10001000	46	10101111	47	00111001
48	10100010	49	10110000	50	01011101	51	10111110	52	10000100
53	11001111	54	10111111	55	10101001	56	11001010	57	11010000
58	11010001	59	11011111	60	11101001	61	11110100	62	11111101
63	1010011	64	0011001	65	0111110	66	0100110	67	0111100
68	0011000	69	0111000	70	0000100	71	0101100	72	0010100
73	0101101	74	0001001	75	11101011	76	11111111	77	11101111
78	11100000	79	11110111	80	11101010	81	11111100	82	11011001
83	0000011	84	11100001	85	11110000	86	0001000	87	0001111
88	0010011	89	0110010	90	0011101	91	1000001	92	1100011
93	0111001	94	1001010	95	0111101	96	1100010	97	1001110
98	0001101	99	1011100	100	0110101	101	0110011	102	0010101
103	0010110	104	0001011	105	0100011	106	11100101	107	11101101
108	0100000	109	11110011	110	0001100	111	11101100	112	0100111
113	0101000	114	11110001	115	0110100	116	0010010	117	11111010
118	11111000	119	11100111	120	0011010	121	0101010	122	11100110
123	0011110	124	1000110	125	1000111	126	0101111	127	1011011
128	0101001	129	1010110	130	0100001	131	1100001	132	1000011
133	1010000	134	1010101	135	1010010	136	1001100	137	1001011
138	0011111	139	11111001	140	0100100	141	11110101	142	11011010
143	11100011	144	11001101	145	10110100	146	10101000	147	11010010
148	01101100	149	10111010	150	01101110	151	01110110	152	10110101
153	10000000	154	00000001	155	00010101	156	00111000	157	111110111
158	00101110	159	00001011	160	110110110	161	110101101	162	101110110
163	110110000	164	011011011	165	101011101	166	011101000	167	101110111
168	001011110	169	101011100	170	001011111	171	011101001	172	100110111
173	100001011	174	101000110	175	011011010	176	110100110	177	110010111
178	100001010	179	110100111	180	110110001	181	111000101	182	110101100
183	111000100	184	01001010	185	00010100	186	111100100	187	111011100
188	111010000	189	111100101	190	110111000	191	111001001	192	00100001
193	111010001	194	110011100	195	00000100	196	111101101	197	111110110
198	00001010	199	111101100	200	00011100	201	01010111	202	00000101
203	01011100	204	01101111	205	00000000	206	110110111	207	111001000
208	110101000	209	010010111	210	1101010011	211	000111011	212	1100101101
213	1100111010	214	1101110010	215	11001110111	216	11001011001	217	0001110100
218	00011101010	219	11001011000	220	110111001110	221	110111001111	222	000111010111
223	100110110010	224	000111010110	225	110011101100	226	11001110110110	227	110011101101110
228	110011101101111	229	1100111011010000	230	110011101101001	231	110011101101010	242	1100111011010001

Table 6: Char Code Dictionary for $image.dat$

char	code	char	code	char	code	char	code	char	code
10	110101	32	111	33	1100001010000	34	11000000110	39	00101010
44	0110101	45	110000000	46	011011	48	11000011101	49	001010110
50	11000010110	51	001010111100	52	001010111101	53	001010111110	54	001010111111
55	110000001000	56	110000001001	57	110000001010	58	1100001010001	63	1100001010010
65	0010100010	66	11000010111	67	00101001100	68	00101001101	69	1100001010011
70	00101001110	71	00101001111	72	11000011000	73	11000001	75	110000001011
76	0010100011	77	00101000000	78	1100001111	79	11000000111	80	00101000001
83	1100001101	84	110000100	86	1100001010100	87	0010101110	89	11000011001
97	0111	98	1101101	99	001011	100	110111	101	000
102	100000	103	00100	104	10011	105	0101	106	001010010
107	0110100	108	10010	109	110001	110	0100	111	1011
112	100001	113	00101000010	114	11001	115	0011	116	1010
117	10001	118	1101100	119	110100	120	11000011100	121	01100
122	00101000011	128	1100001010101	148	1100001010110	226	1100001010111		

Table 7: Char Code Dictionary for $text.dat$