Information theory: HW #3 solution

Table of Contents

1 Task objectives, format, prerequisites

So the task is to implement several algorithms of data compression and provide reports on them. Reports will contain logs from algorithms that will show how it operates this data. Tables with explanation will be attached. I won't attach haskell snippets this time as they became really big and complex. Instead, i'll put it into another pdf and send it with this one.

Proverb correspondent to my task is the following:

> *Love the heart that hurts you, but never hurt the heart that loves you.*

It's in english that's why I'll stick to utf-8/ascii encoding as byte representation. It contains 71 letters, so it takes 568 bytes if used without any coding.

Remark: sorry, but my rendering format doesn't support proper underscore marks (_) – they turn into underlines and i see no way to escape them. "␣" can be used instead, but it's too ugly. I'll use it when ambiguous, and just leave empty place in other places. Yes, and sorry for the tables. They're ugly both when they're are and are not full-width. Rule of thumb: use the same table formatting as in the textbook. So I choose to use full width.

2 Two-pass huffman coding

I've implemented regural huffman coding, as it can be seen from the first table: code words of the same length are sorted lexicographically:

$Char$	$Codeword$	$Frequency$
	00	13
e	010	8
h	011	8
t	100	11
r	1010	5
u	1011	5
s	11000	2
y	11001	2
v	11010	3
a	11011	4
o	11100	4
,	111010	1
.	111011	1

l	111100	1
n	111101	1
L	111110	1
b	111111	1

And the second pass completes encoding:

C	$P(C)$	$Codeword$	$L(S)$
L	1 / 71	111110	6
o	4 / 71	11100	11
v	3 / 71	11010	16
e	8 / 71	010	19
	13 / 71	00	21
t	11 / 71	100	24
h	8 / 71	011	27
e	8 / 71	010	30
	13 / 71	00	32
h	8 / 71	011	35
e	8 / 71	010	38
a	4 / 71	11011	43
r	5 / 71	1010	47
t	11 / 71	100	50
	13 / 71	00	52
t	11 / 71	100	55
h	8 / 71	011	58
a	4 / 71	11011	63
t	11 / 71	100	66
	13 / 71	00	68
h	8 / 71	011	71
u	5 / 71	1011	75
r	5 / 71	1010	79
t	11 / 71	100	82
s	2 / 71	11000	87
	13 / 71	00	89
y	2 / 71	11001	94
o	4 / 71	11100	99
u	5 / 71	1011	103
,	1 / 71	111010	109
	13 / 71	00	111
b	1 / 71	111111	117
u	5 / 71	1011	121
t	11 / 71	100	124
	13 / 71	00	126
n	1 / 71	111101	132
e	8 / 71	010	135
v	3 / 71	11010	140
e	8 / 71	010	143
r	5 / 71	1010	147
	13 / 71	00	149
h	8 / 71	011	152
u	5 / 71	1011	156
r	5 / 71	1010	160
t	11 / 71	100	163

	13 / 71	00	165
t	11 / 71	100	168
h	8 / 71	011	171
e	8 / 71	010	174
	13 / 71	00	176
h	8 / 71	011	179
e	8 / 71	010	182
a	4 / 71	11011	187
r	5 / 71	1010	191
t	11 / 71	100	194
	13 / 71	00	196
t	11 / 71	100	199
h	8 / 71	011	202
a	4 / 71	11011	207
t	11 / 71	100	210
	13 / 71	00	212
l	1 / 71	111100	218
o	4 / 71	11100	223
v	3 / 71	11010	228
e	8 / 71	010	231
s	2 / 71	11000	236
	13 / 71	00	238
y	2 / 71	11001	243
o	4 / 71	11100	248
u	5 / 71	1011	252
.	1 / 71	111011	258

Encoded proverb encoded is $l_2 = 258$ bits, while data amount needed to transfer huffman tree should be calculated manually. First, to transfer the tree itself, it's sufficient to pass only 19 bits (calculated manually using table 3.3 from the study book).

$$\left\lceil \log \binom{256}{1} \right\rceil + \left\lceil \log \binom{255}{3} \right\rceil + \left\lceil \log \binom{254}{2} \right\rceil + \left\lceil \log \binom{253}{5} \right\rceil + \left\lceil \log \binom{252}{6} \right\rceil = 117$$

So totally $l_1 = 19 + 117 = 136$, and total length is $l = l_1 + l_2 = 394$ bits. Much better than raw 568 bits.

3 Adaptive arithmetic coding

Implementation uses renormalization together with fixed-point precision arithmetics (16 bits words). Algorithm "A" from textbook is used (has $n + 1$ in denominator).

Here is the table algorithm outputs (backslash stands for escape symbol):

C	$P(C)$	$Codeword$	$L(S)$
esc	1 / 1		0
L	1 / 256	0100110	7
esc	1 / 2		7
o	1 / 255	00110111	15
esc	1 / 3		15
v	1 / 254	0001010	22
esc	1 / 4	00010	27
e	1 / 253	0100101	34
esc	1 / 5	010	37

	1 / 252	00111001	45
~~esc~~	~~1 / 6~~	~~100~~	~~48~~
t	1 / 251	0101110	55
esc	1 / 7	01	57
h	1 / 250	000110001	66
e	1 / 8	001	69
	1 / 9	00	71
h	1 / 10	0	72
e	2 / 11	0000	76
esc	1 / 12	0	77
a	1 / 249	000001000	86
esc	1 / 13	00011	91
r	1 / 248	00100000	99
t	1 / 14	0	100
	2 / 15	000000	106
t	1 / 8	010	109
h	2 / 17	001	112
a	1 / 18	0001	116
t	3 / 19	0	117
	3 / 20	00000	122
h	1 / 7	0	123
esc	1 / 22	001000	129
u	1 / 247	0010110	136
r	1 / 23	00100	141
t	1 / 6	0	142
esc	1 / 25	0001	146
s	1 / 246	0000011	153
	2 / 13	0000000	160
esc	1 / 27	101	163
y	1 / 245	000101111	172
o	1 / 28	00111	177
u	1 / 29	011	180
esc	1 / 30	0010001	187
,	1 / 244	00011011	195
	5 / 31	00	197
esc	1 / 32	011001	203
b	1 / 243	010	206
u	2 / 33	000001	212
t	5 / 34	00010	217
	6 / 35	000	220
esc	1 / 36	101	223
n	1 / 242	0001000011	233
e	3 / 37	0	234
v	1 / 38	000110	240
e	4 / 39	000	243
r	1 / 20	0001	247
	7 / 41	00000	252
h	2 / 21	010	255
u	3 / 43	011	258
r	3 / 44	001	261
t	2 / 15	00	263
	4 / 23	00000	268
t	7 / 47	01	270
h	5 / 48	00	272

e	5 / 49	000010	278
	9 / 50	00	280
h	2 / 17	0	281
e	3 / 26	00001	286
a	2 / 53	00	288
r	2 / 27	00001	293
t	8 / 55	00	295
	5 / 28	00000	300
t	3 / 19	011	303
h	7 / 58	01	305
a	3 / 59	0001	309
t	1 / 6	0	310
	11 / 61	00000	315
esc	1 / 62	100011	321
l	1 / 241	001111	327
o	2 / 63	0001011	334
v	1 / 32	1001	338
e	7 / 65	00	340
s	1 / 66	0001	344
	12 / 67	0000000	351
y	1 / 68	1	352
o	1 / 23	000000110	361
u	2 / 35	100	364
esc	1 / 71	0100100	371
.	1 / 240	0001100	378
final		1000111	385

On every step algorithm saves 16-digit high and low variables. If the interval we go into is small enough that new bounds share most significant bits, we put them on the wire. This algorithm is decribed both in textbook and on wikipedia[1] and also in Amir Said's article "Introduction to Arithmetic Coding"[2]. Thus empty spaces in third column mean that interval didn't satisfy this property. So eventually it took 385 bits to encode the proverb, better than huffman.

4 Enumerative

First of all I emphasize that no *real encoder* will be implemented, i'll just present here an estimate on how much information will it take. Enumerative encoding implementation seems complex and impractical to do.

Here's the main function that calculates length of the input.

```
enumerative :: BS.ByteString -> Integer
enumerative input = l1 + l2
  where
    n = fromIntegral $ BS.length input
    chars = BS.unpack input
    unique = nub chars
    occurences =
        M.fromList $
        map (\i -> (i, fromIntegral $ length $ filter (== i) chars)) unique
    comp, compcomp, comp' :: [Integer]
    comp = reverse $
          sort $ map (\i -> fromMaybe 0 $ M.lookup i occurences) [0 .. 0xff]
    m = length comp
    compcomp = map (fromIntegral . length) $ group comp
    comp' = filter (> 0) comp
    l2 = ceiling $
```

```
        log2' $ foldr (\x acc -> acc `div` (factorial x)) (factorial n) comp'
    l11 = ceiling $ log2' $ n * product comp'
    l12 = ceiling $
          log2' $
          foldr (\x acc -> acc `div` (factorial x))
                (factorial $ fromIntegral $ length comp)
                compcomp
    l1 = l11 + l12
```

First sorted composition: $\tau = (13, 11, 8, 8, 5, 5, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1, 1, 0, 0, . . , 0, 0)$. Composition of composition $\tau' = (1, 1, 2, 2, 2, 1, 2, 6, 239)$. Length of the composition $l_1 = 154$, number of the proverb in list of strings with this composition $l_2 = 223$. Total information needed to transmit the string: $l = l_1 + l_2 = 377$ bits. A little bit less then with arithmetic coding, less then huffman.

5 LZ77

Implemented version of LZ77 uses levenshtein's code described in textbook (because elias and unary universal codes are less efficient for current dataset). It uses window of size 100, more than proverb's length.

$Flag$	$Substring$	d	l	$Codeword$	$Bits$	$Total$
0	L		0	001001100	9	9
0	o		0	001101111	9	18
0	v		0	001110110	9	27
0	e		0	001100101	9	36
0	␣		0	001011111	9	45
0	t		0	001110100	9	54
0	h		0	001101000	9	63
1	e␣	4	2	1100100	7	70
1	he	3	2	10011100	8	78
0	a		0	001100001	9	87
0	r		0	001110010	9	96
1	t	8	1	110000	6	102
1	␣th	10	3	11010101	8	110
1	a	6	1	1001100	7	117
1	t␣	5	2	100101100	9	126
1	h	14	1	1011100	7	133
0	u		0	001110101	9	142
1	rt	10	2	101010100	9	151
0	s		0	001110011	9	160
1	␣	21	1	1101010	7	167
0	y		0	001111001	9	176
1	o	26	1	1110100	7	183
1	u	7	1	1001110	7	190
0	,		0	000101100	9	199
1	␣	26	1	1110100	7	206
0	b		0	001100010	9	215
1	u	11	1	10010110	8	223
1	t␣	20	2	1010100100	10	233
0	n		0	001101110	9	242
1	e	33	1	11000010	8	250
1	ve	35	2	1100011100	10	260
1	r	27	1	10110110	8	268
1	␣hurt	21	5	101010111001	12	280
1	␣the␣heart␣that␣	41	16	1101001111100000	16	296

0	l		0	001101100	9	305
1	ove	61	3	1111101101	10	315
1	s␣you	41	5	1010100111001	13	328
0	.		0	000101110	9	337

Here's also results for other universal codes (smaller windows affect length dramatically because of that "the heart that" chunk in the end. Best performance of levenshtein is achieved because its encoding of "1" takes only 1 bit (compared to 2 bits of elias) and it's more effective then unary on bigger numbers. In general i expect elias to perform better.

$Code$	W	L
Unary	45	344
Unary	50	344
Unary	55	344
Unary	60	344
Unary	65	338
Unary	70	338
Unary	75	338
Levenshtein	45	344
Levenshtein	50	344
Levenshtein	55	344
Levenshtein	60	344
Levenshtein	65	337
Levenshtein	70	337
Levenshtein	75	337
Elias	45	358
Elias	50	358
Elias	55	358
Elias	60	358
Elias	65	350
Elias	70	350
Elias	75	350

So in conclusion we've achieved $l = 337$ bits, which is the best result among experiments for now.

6 LZW

I've implemented LZW algorithm with escape symbol and matched it with the test proverb (if we cannot…), got 291 bit as in the textbook. Here are the result on the real proverb:

$Dictionary$	$Match$	$Dict\ index$	$Codeword$	$Bits$	$Total$
L		0	01001100	8	8
o		0	01101111	8	16
v		0	001110110	9	25
e		0	0001100101	10	35
␣		0	0001011111	10	45
t		0	00001110100	11	56
h		0	00001101000	11	67
e␣	e	4	100	3	70
␣h	␣	5	101	3	73
he	h	7	0111	4	77
ea	e	4	0100	4	81
a		0	000001100001	12	93

r		0	000001110010	12	105
t␣	t	6	0110	4	109
␣t	␣	5	0101	4	113
th	t	6	0110	4	117
ha	h	7	0111	4	121
at	a	12	01100	5	126
t␣h	t␣	14	01110	5	131
hu	h	7	00111	5	136
u		0	0000001110101	13	149
rt	r	13	01101	5	154
ts	t	6	00110	5	159
s		0	0000001110011	13	172
␣y	␣	5	00101	5	177
y		0	0000001111001	13	190
ou	o	2	00010	5	195
u,	u	21	10101	5	200
,		0	0000000101100	13	213
␣b	␣	5	00101	5	218
b		0	0000001100010	13	231
ut	u	21	10101	5	236
t␣n	t␣	14	01110	5	241
n		0	00000001101110	14	255
ev	e	4	000100	6	261
ve	v	3	000011	6	267
er	e	4	000100	6	273
r␣	r	13	001101	6	279
␣hu	␣h	9	001001	6	285
ur	u	21	010101	6	291
rt␣	rt	22	010110	6	297
␣th	␣t	15	001111	6	303
he␣	he	10	001010	6	309
␣he	␣h	9	001001	6	315
ear	ea	11	001011	6	321
rt␣t	rt␣	41	101001	6	327
tha	th	16	010000	6	333
at␣	at	18	010010	6	339
␣l	␣	5	000101	6	345
l		0	00000001101100	14	359
ov	o	2	000010	6	365
ves	ve	36	100100	6	371
s␣	s	24	011000	6	377
␣yo	␣y	25	011001	6	383
ou.	ou	27	011011	6	389
.		0	00000000101110	14	403

Well, results ($l = 403$ bits) are clearly worse than they were with previous coding algorithms. On the contrary, LZW implementation is pretty simple and straight-forward. I assume that the biggest problem of LZW here is the big variety of new symbols (rather big) comparing to the length of input data – we've spent a lot of bits to transmit new characters, especially coding escape symbol. Should perform better then lz77 on bigger datasets.

For comparison i've taken this phrase that's 74 words and 500 bytes of lorem ipsum text:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras ornare diam nec interdum mollis. Phasellus tortor felis, dapibus eu bibendum eu, commodo quis erat. Vestibulum fringilla, purus semper eleifend laoreet, sem dui volutpat lectus, sed ullamcorper ante neque id lectus. Nulla ullamcorper egestas nisl, at convallis leo tempus vel. Sed mi lacus, aliquam ullamcorper purus vitae, vulputate dignissim ipsum. Nam in est eu quam maximus blandit. Integer nec iaculis felis. Vestibulum ut cras amet.

Here's a comparison of LZW/LZ77 with different window sizes and universal codes. Maximum windows size is 4000 (500 bytes):

$Algorithm$	W	$UniversalCode$	$Totalbits$
LZW	∅	∅	2617
LZ77	Unary	500	2761
LZ77	Unary	1000	2761
LZ77	Unary	2000	2761
LZ77	Unary	4000	2761
LZ77	Levenshtein	500	2835
LZ77	Levenshtein	1000	2835
LZ77	Levenshtein	2000	2835
LZ77	Levenshtein	4000	2835
LZ77	Unary	200	2932
LZ77	Elias	500	2949
LZ77	Elias	1000	2949
LZ77	Elias	2000	2949
LZ77	Elias	4000	2949
LZ77	Levenshtein	200	3021
LZ77	Unary	100	3048
LZ77	Levenshtein	100	3132
LZ77	Elias	200	3148
LZ77	Unary	50	3283
LZ77	Elias	100	3305
LZ77	Levenshtein	50	3346
LZ77	Elias	50	3591

Interestingly, unary universal coding performs on average better than levenshtein/elias. And most importantly, our assumption appeared to be correct – LZW saves us 144 bits (0.34 percent).

7 PPM

I've chosen PPMA as main algorithm to implement because it's simpler to debug. I've added support of exceptions. First thing I need to say is that table 4.5 of the textbook has a mistake. On step 39 $p(a|s)$ is calculated incorrectly: initial context s `OULD_` is matched once before on the step 24 (as it's written in $\tau_t(s)$ of step 39), then it's reduced three times to _. Notice that no bigger suffix of `OULD_` (then _) is matched in other places before – there's only one `D_` at string's position 27. So $\tau_t(s)$ of step 39 says that _ is found 8 times. Indeed – on positions $3, 6, 13, 16, 19, 22, 28, 31$. Using exceptions rule we ignore all _ that are followed by characters that followed any of our pre-contexts: one of `OULD_`, `ULD_`, `LD_`, `D_`. But that's exactly only one character `W`. So positions $6, 13, 17, 31$ are left (and on position 13 _ is followed by `D`), which gives us a probability of `D` after _:

$$p_t(a|s) = \frac{\tau_t(s,a)}{\tau_t(s)+1} = \frac{1}{4+1} = \frac{1}{5}$$

Textbook says that correct probability is $\frac{1}{4}$. So that's a mistake in my opinion. The correct length of coded string is this 249 bits, not 250.

Back to the task proverb. Here's a table with algorithm trace, launched with $D = 5$:

$Char$	$Context\ s$	$\tau_t(s)$	$p_t(esc \mid s)$	$p_t(a \mid s)$
L	#	0	1/1	1/256
o	#	1	1/2	1/255
v	#	2	1/3	1/254
e	#	3	1/4	1/253
␣	#	4	1/5	1/252
t	#	5	1/6	1/251
h	#	6	1/7	1/250
e	#	7		1/8
␣	"e"	1		1/2
h	"e␣"	1,1,9	1/2,1/1	1/9
e	"h"	1		1/2
a	"he"	1,2,11	1/2,1/1,1/10	1/249
r	#	12	1/13	1/248
t	#	13		1/14
␣	"t"	1,14	1/2	2/13
t	"␣"	2		1/3
h	"␣t"	1		1/2
a	"␣th"	1,1,2,17	1/2,1/1,1/1	1/15
t	"a"	1,18	1/2	1/6
␣	"t"	3		1/4
h	"t␣"	1,3	1/2	1/2
u	"␣h"	1,3,21	1/2,1/2,1/17	1/247
r	#	22		1/23
t	"r"	1		1/2
s	"rt"	1,4,24	1/2,1/3,1/17	1/246
␣	#	25		2/13
y	"␣"	4,26	1/5,1/18	1/245
o	#	27		1/28
u	"o"	1,28	1/2	1/28
,	"u"	1,29	1/2,1/28	1/244
␣	#	30		5/31
b	"␣"	5,31	1/6,1/22	1/243
u	#	32		2/33
t	"u"	2,33	1/3	5/31
␣	"t"	5		1/3
n	"t␣"	2,6,35	1/3,1/3,1/24	1/242
e	#	36		3/37
v	"e"	3,37	1/4	1/29
e	"v"	1		1/2
r	"ve"	1,4,39	1/2,1/3	2/29
␣	"r"	2,40	1/3	1/5
h	"␣"	7		1/4
u	"␣h"	2		1/3
r	"␣hu"	1		1/2
t	"␣hur"	1		1/2
␣	"␣hurt"	1,1,1,2	1/2,1/1,1/1	1/2
t	"rt␣"	1		1/2
h	"rt␣t"	1		1/2
e	"rt␣th"	1,1,2	1/2,1/1	1/2
␣	"␣the"	1		1/2
h	"␣the␣"	1		1/2

e	"the␣h"	1		1/2
a	"he␣he"	1		1/2
r	"e␣hea"	1		1/2
t	"␣hear"	1		1/2
␣	"heart"	1		1/2
t	"eart␣"	1		1/2
h	"art␣t"	1		1/2
a	"rt␣th"	2		1/3
t	"t␣tha"	1		1/2
␣	"␣that"	1		1/2
l	"that␣"	1,1,1,5,11,61	1/2,1/1,1/1,1/5,1/3,1/40	1/241
o	#	62		2/63
v	"o"	2		1/3
e	"ov"	1		1/2
s	"ove"	1,2,7,65	1/2,1/2,1/4	1/42
␣	"s"	1		1/2
y	"s␣"	1		1/2
o	"s␣y"	1		1/2
u	"s␣yo"	1		1/2
.	"s␣you"	1,1,1,1,4,70	1/2,1/1,1/1,1/1,1/4,1/54	1/240

Finaly we have $l = 345 + 1 = 346$ bits. Result is pretty good, much better then previous algorithm implementations. Here's also a comparison table for different D – both original proverb and lorem ipsum text mentioned in the previous section:

D	$Proverb$	$Lorem\ Ipsum$
1	358	2340
2	348	2355
3	345	2364
4	345	2367
5	346	2366
6	345	2366
7	345	2366
8	345	2366
9	345	2366
10	345	2366
20	345	2366

Thus we can see that in general big window D doesn't help with english text, because on average coincedences of big words (>5 symbols) are pretty rare.

8 Burrows-Wheeler + book stack

Implemented algorithm uses Burrows-Wheeler transformation with MTF (book stack) algorithm together to obtain results. That's a first round, straight-forward MTF without escapes.

One thing to mention before is that I'm encoding $diff + 1$ where $diff$ is a number of different words between current and previous occurences of the char because universal coding work only in range 1.., whereas $diff$ can be zero (example – `aaaa`). Used universal coding is monotonic (levenshtein). When character is not in the history, i'm assuming there's a list of all ascii characters (length 256) before the processed string.

First, let's perform the transformation. Here's the result:

```
uu.,eerttttetsseehh_hhvhhvvntttt______yyLleaauuteaaurrr____roohhboeo__
```

And the number of this string in sorted list is 3 (index 2). Here's a table that shows how algorithm worked:

Char	*New*	*Dist*	*Diff*	*Code word*	*Bits*	*Total*
u	1	139	139	1110110001100	13	13
u	0	2	0	10	2	15
.	1	212	211	1110111010100	13	28
,	1	215	214	1110111010111	13	41
e	1	159	158	1110110011111	13	54
e	0	2	0	10	2	56
r	1	148	146	1110110010011	13	69
t	1	147	145	1110110010010	13	82
t	0	2	0	10	2	84
t	0	2	0	10	2	86
t	0	2	0	10	2	88
t	0	2	0	10	2	90
e	0	8	2	101	3	93
t	0	3	1	100	3	96
s	1	155	147	1110110010100	13	109
s	0	2	0	10	2	111
e	0	5	2	101	3	114
e	0	2	0	10	2	116
h	1	170	159	1110110100000	13	129
h	0	2	0	10	2	131
	1	181	169	1110110101010	13	144
h	0	3	1	100	3	147
h	0	2	0	10	2	149
v	1	161	147	1110110010100	13	162
h	0	3	1	100	3	165
h	0	2	0	10	2	167
v	0	4	1	100	3	170
v	0	2	0	10	2	172
n	1	174	156	1110110011101	13	185
t	0	17	6	110011	6	191
t	0	2	0	10	2	193
t	0	2	0	10	2	195
t	0	2	0	10	2	197
	0	14	4	110001	6	203
	0	2	0	10	2	205
	0	2	0	10	2	207
	0	2	0	10	2	209
	0	2	0	10	2	211
	0	2	0	10	2	213
y	1	174	146	1110110010011	13	226
y	0	2	0	10	2	228
L	1	221	192	1110111000001	13	241
l	1	190	161	1110110100010	13	254
e	0	27	8	1101001	7	261
a	1	203	173	1110110101110	13	274
a	0	2	0	10	2	276
u	0	46	14	1101111	7	283
u	0	2	0	10	2	285
t	0	17	7	1101000	7	292
e	0	7	3	110000	6	298

a	0	6	3	110000	6	304
a	0	2	0	10	2	306
u	0	6	3	110000	6	312
r	0	48	12	1101101	7	319
r	0	2	0	10	2	321
r	0	2	0	10	2	323
	0	19	8	1101001	7	330
	0	2	0	10	2	332
	0	2	0	10	2	334
	0	2	0	10	2	336
r	0	6	1	100	3	339
o	1	206	160	1110110100001	13	352
o	0	2	0	10	2	354
h	0	39	12	1101101	7	361
h	0	2	0	10	2	363
b	1	223	174	1110110101111	13	376
o	0	5	2	101	3	379
e	0	19	7	1101000	7	386
o	0	3	1	100	3	389
	0	11	5	110010	6	395
	0	2	0	10	2	397

So we get 393 bits in total for transmitting this sequence using straight-forward MTF method. Also index 3 from BWT will take 3 bits if encoded using elias coding. So in total 396 bits. Already better than raw 568, but worse than algorithms mentioned before. Performance with other universal codes is worse – 400 in total with elias and 2975 in total with unary (no wonder, encoding big numbers takes a lot).

Let's take another approach with escape symbols and enumerative coding which is described in part 4.6 of the textbook.

Running mtfs transfromation will yield this result:

```
esc 0 esc esc esc 0 esc esc 0 0 0 0 2 1 esc 0 2 0 esc 0 esc 1 0 esc 1 0 1 0 esc
6 0 0 0 4 0 0 0 0 0 esc 0 esc esc 8 esc 0 14 0 7 3 3 0 3 12 0 0 8 0 0 0 1 esc 0
12 0 esc 2 7 1 5 0
```

Coding it using enumerative coding gives us 267 bits (i just ran a function `enumerative`, it doesn't trace anything). Transferring a character per escape with 17 escapes in total gives us $17 \times 8 = 136$ bits. Summing up, we get $267 + 136 + 3 = 406$ (remember 3 bits from BWT). Less than with straight-forward MTF coding. Both approaches are pretty simple to implement on the other hand.

9 Standart archiving

Last step is try to apply some standart archiving functions to our proverb. I'll be really simple here and try it out with `GZip` algorithm from library `zlib`[3]. It uses "Deflate" algorithm that's a combination of LZ77 and Huffman coding[4].

So results are pretty sad for the current proverb: $l = 544$, only slightly lower than original 568 bits. On the other hand, lorem ipsum text is compressed from 4000 to 2472 bits, which is a significant performance boost. Small texts are not that representative anyways.

10 Comparison and summary

Let's compare all approaches we used:

Method	*Bits*
Plain	568
Two pass huffman	394
Adaptive arithmetic	385
Enumerative	377
LZ77	337
LZW	403
PPMA	346
BW+MTF plain	396
BW+MTF enumerative	406
GZip	544

It's hard to say which one is better in general, but in our case LZ77 is as absolute winner (though we used a large window which is not suitable for real usage), PPMA is the second one (more or lessfair). GZip showed the worst together with LZW. We don't compare speed efficiency of algorithms which is important parameter, comparing compression rate doesn't really make the sense in general – different tasks define different requirements for encoding algorithms. Another conclusion (that was covered in textbook too) is that two-pass algorithms are not that much better comparing to one-pass and it's clearly true based on our comparison table.

Footnotes:

[1] https://en.wikipedia.org/wiki/Arithmetic_coding#Precision_and_renormalization

[2] http://www.hpl.hp.com/techreports/2004/HPL-2004-76.pdf

[3] https://hackage.haskell.org/package/zlib-0.6.1.1/docs/Codec-Compression-GZip.html

[4] http://zlib.net/

Author: Volkhov Mikhail, M3338
Created: 2016-12-18 Sun 17:13