Shox++ - Guaranteed Compression of Short Strings

Arundale Ramanathan

August 4, 2019

Abstract

We formulate a hybrid encoding method with which short strings could be compressed using context aware pre-mapped codes resulting in surprisingly good ratios.

We also go on to prove that this technique can guarantee compression for any English language sentence of minimum 3 words.

1 Summary

Compression of Short Strings of arbitrary lengths have not been addressed sufficiently by lossless entropy encoding methods so far. Although it appears inconsequential, space occupied by such strings become significant in memory constrained environments such as Arduino Uno and when attempting storage of such independent strings in a database. While block compression is available for databases, retrieval efficiency could be improved if the strings are individually compressed.

2 Basic Definitions

In information theory, *entropy encoding* is a lossless data compression scheme that is independent of the specific characteristics of the medium [1].

One of the main types of entropy coding creates and assigns a unique prefix-free code to each unique symbol that occurs in the input. These entropy encoders then compress data by replacing each fixed-length input symbol with the corresponding variable-length prefix-free output codeword.

According to Shannon's source coding theorem, the optimal code length for a symbol is $-log_b P$, where b is the number of symbols used to make output codes and P is the probability of the input symbol [2]. Therefore, the most common symbols use the shortest codes.

The most popular and most used method (even today) for forming optimal prefix-free discrete codes is Huffman coding [3].

In contrast to entropy encoding, there are various other approaches to lossless coding including Lempel-Ziv coding [4] and Burrows-Wheeler coding [5].

3 Existing techniques - Smaz and shoco

While technologies such as GZip, Deflate, Zip, LZMA are available for file compression, they do not provide optimal compression for short strings. Eventhough these methods compress far more than what we are proposing, these methods often expand the original source for short strings because the symbol-code mapping also needs to be attached to aid decompression.

To our knowledge, only two other competing technologies exist - Smaz and shoco.

Smaz is a simple compression library suitable for compressing very short strings [9]. It was developed by Salvatore Sanfilippo and is released under the BSD license.

Shoco is a C library to compress short strings [10]. It was developed by Christian Schramm and is released under the MIT license.

While both are lossless encoding methods, Smaz is dictionary based and Shoco classifies as an entropy coder [10].

In addition to providing a default frequency table as model, shoco provides an option to re-define the frequency table based on training text [10].

4 This research

We propose a hybrid encoding method which mainly relies on entropy encoding method for compression.

Unlike shoco, we propose a fixed frequency table generated based on the characterestics of English language letter frequency. We re-use the research carried out by Oxford University [7] and other sources [6] [8] and come out with a unique method that takes advantage of the conventions of the language.

We propose a single model that presently is fixed because of the advantages it offers over the training models of shoco.

The disadvantage with the training model, although it may appear to offer more compression, is that it does not consider the patterns that usually appear during text formation.

We can actually see that this performs better than pre-trained model of shoco (See performance section).

For supporting other languages and types of text, we propose to use different models tailored to each of them, considering the patterns of the specific language.

Unlike smaz and shoco, we assume no *a priori* knowledge about the input text. However we rely on *a posteriori* knowledge about the research carried out on the language and common patterns of sentence formation and come out with pre-assigned codes for each letter.

5 Model

In the ASCII chart, we have 95 printable letters starting from 32 through 126. For the purpose of arriving at fixed codes for each of these letters, we use two sets of prefix-free codes.

The first set consists of 11 codes, which are: 00, 010, 011, 100, 1010, 1011, 1100, 1101, 1110, 11110, 11111. The second set consists of 7 codes, which are 0, 10, 110, 11100, 11101, 11110, 11111.

With these two sets of codes, we form several sets of letters as shown in the table below and use some rules based on how patterns appear in short strings, to arrive at frequency table.

hcode →	**10**	**0**	**110**	**11100**	**11101**	**11110**	**11111**
↓ vcode	**Set 1**	**Set 1a**	**Set 1b**	**Set 2**	**Set 3**	**Set 4**	**Set 4a**
00	switch	l / L	y / Y	switch	.	&	@
010	sp / tb	c / C	v / V	9	,	;	?
011	e / E	d / D	k / K	0	-	:	'
100	Uni / Cont.	h / H	q / Q	1	/	<	^
1010	t / T	u / U	j / J	2	=	>	#
1011	a / A	p / P	x / X	3	+	*	_
1100	o / O	m / M	z / Z	4	sp	"	!
1101	i / I	b / B	cr+lf / rpt	5	(	{	\
1110	n / N	g / G	lf / bin	6	)	}	|
11110	s / S	w / W	dict /	7	$	[	~
11111	r / R	f / F	term / cr	8	%	]	`

6 Rules

6.1 Basic rules

- It can be seen that the more frequent symbols are assigned smaller codes.
- Set 1 is always active when beginning compression. So the letter e has the code 011, t 100 and so on.
- If the letter in Set 1a needs to be encoded, the switch code is used followed by 0 to indicate Set 1a. So the letter c is encoded as 00000, d as 000010 and so on.
- Similarly, if the letter in Set 1b needs to be encoded, the switch code is used followed by 110. So k is encoded as 00110010, q as 00110011 and so on. Note that the terminator symbol is encoded as 0011011111.

6.2 Upper case symbols

- For encoding uppercase letters in Set 1, the switch symbol is used followed by 10 and the code against the symbol itself. For example, E is encoded as 0010011. The same applies to tab character.
- For encoding uppercase letters in Set 1a, the switch symbol is used, followed by 10, again switch symbol, followed by 0 and then the corresponding code against the letter. For instance, the letter P is encoded as 00100001010.
- Similarly, for uppercase letters in Set 1b, the prefix 001000110 is used. So the symbol X is encoded as 0010001101010.
- If uppercase letters appear continuously, then the encoder may decide to switch to upper case using the prefix 00100010. After that, the same codes for lower case are used to indicate upper case letters until the code sequence 0010 is used again to return to lower case.

6.3 Numbers and related symbols

- Symbols in Set 2 are encoded by first switching to the set by using 00 followed by 11100. So the symbol 9 is encoded as 0011100010.
- For Set 2, whenever is switch is made from Set 1, it makes Set 2 active. So subsequent numbers are encoded without the switch symbol, as in 1011 for 3, 1100 for 4 and so on.
- To return to Set 1, the prefix 0010 is used.
- To encode symbols in Set3, the prefix 00 is used followed by 0 and the corresponding code for the symbol. For example, + is encoded as 0001011.

6.4 Other symbols (Set 4)

- The special characters in Set 4 can be encoded by using the prefix 0011110 followed by the corresponding code for the letter. Example: ; is encoded as 0011110010.
- The symbols in Set 4a are encoded using the prefix 0011111. Example: ? is encoded as 0011111010.

6.5 Sticky sets

- When switching to Set 2, it becomes active and is said to be sticky till Set 1 is made active using the symbol 0010.
- However, no other set is sticky. Set 1 is default. Set 2 automatically becomes sticky when switched to it by using 0011100 and Upper case letters can be made sticky by using 00100010.

- Symbols in Set1a, Set1b, Set3, Set 4 and Set 4a are never sticky. Once encoded the previous sticky set becomes active.

6.6 Special symbols

- term in Set 1b indicates termination of encoding. This is used if length of the encoded string is not stored.
- rpt in Set 1b indicates that the symbol just encoded is to be repeated specified number of times.
- dict in Set 1b indicates that the specified offset in file and length is to be copied at the current position. This is dictionary based encoding.
- CRLF in Set 1a is encoded using a single code. It will be expanded as two bytes CR LF. If only LF is used, such as in Unix like systems, a separate code is used in Set 1a. Also, in the rare case that only CR appears, a longer code is provided in Set1a.

6.7 Dual access for Set 3

- Set 3 can be accessed both when Set 1 and Set 2 is active. This is because the symbols occur commonly in both Set 1 and 2. So it is necessary to have minimum length codes for these.
- For the same reason, the space symbol appears both in Set 1 and Set 3.

6.8 Repeating letters

- If any letter repeats more than 3 times, we use a special code (rpt) shown in Set1b of the model.
- The encoder first codes the letter using the above codes. Then the rpt code is used followed by the number of times the letter repeats.
- The number of times the letter repeats is coded using a special bit sequence as explained in section "Encoding counts" that follows.

6.9 Repeating sections

- If a section repeats, another code of Set1b (dict) is used followed by four fields as described next.
- The first field would be 0 or 1. 0 indicates that the distance of the previous section found repeating is to be counted from beginning. 1 indicates that it is to be counted backwards from current position of encoder.
- The second field indicates the length of the section that repeats.

- The third field indicates the distance of the repeating section.
- The fourth field is coded only if the distance starts from the beginning. It is a number indicating the set that the section belongs to, assuming there are several sets of text are being encoded. If only one set of text is encoded, the first field is coded as 1 indicating the distance is to be counted from the current position.
- The second, third and fourth fields are encoded as explained in the following section "Encoding counts".

6.10 Encoding Counts

- For encoding counts such as length and distance, the horizontal codes shown in the model are re-used, each code indicating how many bits will follow to indicate count.
- If code is 0, 2 bits would follow, that is, count is between 0 and 3.
- If code is 10, 5 bits would follow, that is, count is between 4 and 35.
- If code is 110, 7 bits would follow, that is, count is between 36 and 163.
- If code is 11100, 9 bits would follow, that is, count is between 164 and 675.
- If code is 11101, 12 bits would follow, that is, count is between 676 and 4771.
- If code is 11110, 16 bits would follow, that is, count is between 4772 and 70307.
- If code is 11111, a varint would follow, which means a VarInt that uses a terminating byte would be used to indicate the count. The format for this code is kept for future expansion as it is not expected that an implementation of Shox96 would need this.

Based on the above rules the following Frequency table is formed.

7 Frequency table

ASCII Code	Letter	Code	Length
32		010	3
33	!	00111111100	11
34	“	00111101100	11
35	#	00111111010	11
36	$	001110111110	11
37	%	001110111111	12
38	&	001111000	9
39	’	0011111011	10
40	(	00111011101	11
41	)	00111011110	11
42	*	00111101011	11
43	+	00111011011	11
44	,	0011101010	10
45	-	0011101011	10
46	.	001110100	9
47	/	0011101100	10
48	0	0011100011	10
49	1	0011100100	10
50	2	00111001010	11
51	3	00111001011	11
52	4	00111001100	11
53	5	00111001101	11
54	6	00111001110	11
55	7	001110011110	12
56	8	001110011111	12
57	9	0011100010	10
58	:	0011110011	10
59	;	0011110010	10
60	<	0011110100	10
61	=	00111011010	11
62	>	00111101010	11
63	?	0011111010	10
64	@	001111100	9

ASCII Code	**Letter**	**Code**	**Length**
65	A	00101010	8
66	B	00100001100	11
67	C	001000000	9
68	D	0010000010	10
69	E	0010011	7
70	F	001000011110	12
71	G	00100001101	11
72	H	0010000011	10
73	I	00101100	8
74	J	001000110100	12
75	K	001000110010	12
76	L	001011111	9
77	M	00100001011	11
78	N	00101101	8
79	O	00101011	8
80	P	00100001010	11
81	Q	001000110011	12
82	R	001011110	9
83	S	00101110	8
84	T	0010100	7
85	U	0010000100	10
86	V	00100011000	11
87	W	00100001110	11
88	X	0010001101010	13
89	Y	001000011111	12
90	Z	0010001101011	13
91	[	001111011110	12
92	\	00111111101	11
93	]	001111011111	12
94	ˆ	0011111100	10
95	_	00111111011	11
96	‘	001111111111	12

ASCII Code	Letter	Code	Length
97	a	1010	4
98	b	0001100	7
99	c	00000	5
100	d	000010	6
101	e	011	3
102	f	00011110	8
103	g	0001101	7
104	h	000011	6
105	i	1100	4
106	j	00110100	8
107	k	00110010	8
108	l	11111	5
109	m	0001011	7
110	n	1101	4
111	o	1011	4
112	p	0001010	7
113	q	00110011	8
114	r	11110	5
115	s	1110	4
116	t	100	3
117	u	000100	6
118	v	0011000	7
119	w	0001110	7
120	x	001101010	9
121	y	00011111	8
122	z	001101011	9
123	{	00111101101	11
124	—	00111111110	11
125	}	00111101110	11
126	˜	001111111110	12

Even after the freqency table is formed, the original model is still needed for encoding Sticky sets as explained in the Rules section.

8 Implementation

According to the above Rules and Frequency table, a reference implementation has been developed and made available at https://github.com/siara-cc/Shox96 as shox96_0_2_0.c. This is released under Apache License 2.0.

Further, Shox96 has been used in several open source projects shown below:

- Shox96 Compression Library for Arduino Progmem - https://github.com/siara-cc/Shox96_Arduino_Progmem_lib

- Shox96 Compression Library for Arduino - https://github.com/siara-cc/Shox96_Arduino_lib
- Sqlite3 User Defined Function as loadable extension - https://github.com/siara-cc/Shox96_Sqlite_UDF
- Sqlite3 Library for ESP32 - https://github.com/siara-cc/esp32_arduino_sqlite3_lib
- Sqlite3 Library for ESP8266 - https://github.com/siara-cc/esp_arduino_sqlite3_lib
- Sqlite3 Library for ESP-IDF - https://github.com/siara-cc/esp32-idf-sqlite3

9 Performance Comparison

The compression performance of all three techniques - Smaz, shoco and Shox96 were compared for different types of strings and results are tabulated below:

String	**Length**	**Smaz**	**shoco**	**Shox96**
Hello World	11	10	8	8
The quick brown fox jumps over the lazy dog	43	30	34	29
I would have NEVER said that	28	20	20	18
In (1970-89), $25.9 billion; OPEC bilateral aid [1979-89], $213 million	67	65	52	54

Further - world95.txt - the text file obtained from *The Project Gutenberg Etext of the 1995 CIA World Factbook* was compressed using the three techniques and following are the results:

Original size: 2988577 bytes

After Compression using shoco original model: 2385934 bytes

After Compression using shoco trained using world95.txt: 2088141 bytes

After Compression using Shox96: 1966019 bytes

As for memory requirements, shoco requires over 2k bytes, smaz requires over 1k. But Shox96 requires only around $95 * 3 = 285$ bytes for compressor and decompressor together, ideal for using it with even Arduino Uno.

10 Proving guaranteed compression

Guaranteed compression means that the length of compressed text will never exceed the length of the source text.

While it is not possible to prove it for any text, we can prove this for most real life scenarios good enough for using it without fear of expansion of original length.

At first we make the following assumptions for a given sentence in English language:

- The sentence will start with a capital letter.
- The sentence will end in period (.).
- The sentence will have at least 3 words.
- Special characters other than a-z, A-Z and space will not be more than 2 or 3.
- Terminator symbol is not needed. That is, length of compressed string in bits will be separately maintained.

With the above assumptions, we try to prove guaranteed compression as follows:

- Since the sentence will have atleast two spaces, it saves $5 + 5 = 10$ bits.
- Since any English word will have a vowel and the average length of code in our frequency table is 4, it will save another 12 bits, unless the vowel 'u' appears in all three words, which is not likely in real life.

So, with a saving of atleast 22 bits, we can say it is more than sufficient to offset for any symbol being used, such as Uppercase letter or Special character, provided such letters do not exceed 4, since *the maximum length of any code in our frequency table is only 13.* So if there are 4 such exceeding codes, it will occupy at most $(13 - 8) * 4 = 20$ bits.

This assumption is towards defining a safe limit and since there will be more savings because of the known general frequency of letters, we can safely assume this guarantee.

11 Conclusion

As can be seen from the performance numbers, Shox96 performs better than available techniques. It can also be seen that it performs better for a variety of texts, especially those having a mixture of numbers and special characters.

12 Further work

We propose to improve Shox96 by including further rules for better compression. We also propose to develop such models for other languages and types of text such as Programs.

13 About the Authors

Arundale R. has over 20 years of experience working in the IT industry. He has worked alternatively in large Corporates, MNCs and Startups, including

Viewlocity Asia Pacific Pte. Ltd., IBC Systems Pte. Ltd. and Polaris Software Lab Ltd. He has founded Siara Logics (cc) and Siara Logics (in) and publishing his open source work at https://github.com/siara-cc and https://github.com/siara-in. He has a masters degree in Computer Science from Anna University. He can be reached at arun@siara.cc.

Charumathi A. is a budding Electronics and Communications Engineer studying at Jerusalem College of Engineering, Chennai affiliated to Anna University. She is passionate about building useful technical gadgets, 3d printing, technical research and fashion designing. She can be reached at cm@cm21.in.

Harsha N. is a budding Computer Science Engineer studying at Jerusalem College of Engineering, Chennai affiliated to Anna University. She is passionate about building software projects in C and Java and technical research. She can be reached at nnpharsha562@gmail.com.

References

[1] David MacKay. *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003.

[2] Shannon, Claude E. (July 1948). *"A Mathematical Theory of Communication"*, Bell System Technical Journal. 27

[3] D. A. Huffman, *"A method for the construction of minimum-redundancy codes"*, Proc. IRE, vol. 40, pp. 1098-1101,1952.

[4] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. IEEE Transactions on Information Theory, 23(3):337–343, May 1977.

[5] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Equipment Corporation, Palo Alto, CA, USA, May 1994.

[6] *"Statistical Distributions of English Text"*. data-compression.com. Archived from the original on 2017-09-18.

[7] What is the frequency of the letters of the alphabet in English?, Oxford Dictionary. Oxford University Press. Retrieved 29 December 2012.

[8] Wikipedia, *Letter frequency*, https://en.wikipedia.org/wiki/Letter_frequency, updated December 2018.

[9] Salvatore Sanfilippo, *SMAZ - compression for very small strings*, https://github.com/antirez/smaz, February 2012.

[10] Christian Schramm, *shoco: a fast compressor for short strings*, https://github.com/Ed-von-Schleck/shoco, December 2015.