Unishox 2 - Guaranteed Configurable Compression for Short Strings using Entropy, Dictionary and Delta encoding techniques

Arundale Ramanathan

July 25, 2021

Abstract

Unishox 2 is a hybrid encoding technique replacing Unishox 1, with which short unicode strings could be compressed using context aware pre-mapped codes and delta coding resulting in surprisingly good ratios.

Also shown is how this technique can guarantee compression for any language sentence of minimum 3 words.

1 Summary

Compression of Short Unicode 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 ESP8266 and when attempting storage of such independent strings in a database. While block compression is available for databases, retrieval efficiency could be greatly 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 is about creating and assigning 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 code word.

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 for forming optimal prefix-free discrete codes is Huffman coding [3].

A *Dictionary coder*, also sometimes known as a substitution coder, is a class of lossless data compression algorithms which operate by searching for matches between the text to be compressed and a set of strings contained in a data structure (called the 'dictionary') maintained by the encoder. When the encoder finds such a match, it substitutes a reference to the string's position in the data structure.

The LZ77 family of encoders use the dictionary encoding technique for compressing data. [4]

Delta coding is a technique applied where encoding the difference between the previously encoded symbol or set of symbols is smaller compared to encoding the symbol or the set again. The duffernce is determined by using the set minus operator or subtraction of values. [5]

In contrast to these encoding methods, there are various other approaches to lossless coding including Run Length Encoding (RLE) and Burrows-Wheeler coding [6].

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 [10]. It was developed by Salvatore Sanfilippo and is released under the BSD license.

Shoco is a C library to compress short strings [11]. 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 [11].

In addition to providing a default frequency table as model, shoco provides an option to re-define the frequency table based on training text [11].

4 This research

We propose a hybrid encoding method which relies on the three encoding techniques *viz.* Entropy encoding, Dictionary coding and Delta encoding methods 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 [7] [9] 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 compressing Unicode symbols, we use Delta encoding because usually the difference between subsequent symbols is quite less.

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 27 codes, which are: 00, 010, 011, 1000, 1001, 1010, 1011, 1100, 11010, 11011, 111000, 111001, 111010, 1110110, 1110111, 1111000, 1111001, 1111010, 11110110, 11110111, 11111000, 11111001, 11111010, 11111011, 11111100, 11111101, 11111110, 11111111. We call these as vertical codes (vcodes).

The second set consists of 5 codes, which by default will be 00, 01, 10, 110, 111. We call these horizontal codes (hcodes). These 5 codes can be configured according to the composition of text that needs to be compressed.

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.

hcode →	**00**	**01**	**10**	**110**	**111**
↓ vcode	**Set 1 Alpha**	**Set 2 Sym**	**Set 3 Num**	**Set 4 Dictionary**	**Set 5 Delta**
00	switch	“	switch		
010	sp	{	,		
011	e / E	}	.		
1000	t / T	_	0		
1001	a / A	<	1		
1010	o / O	>	9		
1011	i / I	:	2		
1100	n / N	lf	5		
11010	s / S	crlf	-		
11011	r / R	[	/		
111000	l / L	]	3		
111001	c / C	\	4		
111010	d / D	;	6	<length>	<code>
1110110	h / H	’	7	<distance>	<sign>
1110111	u / U	tab	8		<delta>
1111000	p / P	@	(		
1111001	m / M	*	)		
1111010	b / B	&	sp		
11110110	g / G	?	=		
11110111	w / W	!	+		
11111000	f / F	^	$		
11111001	y / Y	|	%		
11111010	v / V	cr	#		
11111011	k / K		seq4		
11111100	q / Q	‘	seq5		
11111101	j / J	seq1	seq6		
11111110	x / X	seq2	rpt		
11111111	z / Z	seq3	term		

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* 1010 and so on.

6.2 Upper case symbols

- For encoding uppercase letters, the switch symbol is used followed by 00 and the code against the symbol itself. For example, E is encoded as 0000011.

- If uppercase letters appear continuously, then the encoder may decide to switch to upper case using the prefix 0000 0000. After that, the same codes for lower case are used to indicate upper case letters until the code sequence 0000 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 01. So the symbol " is encoded as 00 01 00.
- Numbers in Set 3 are encoded by first switching to the set by using 00 followed by 10. So the symbol 9 is encoded as 00 10 1010.
- For Set 3, whenever is switch is made from Set 1 to any number (0 to 9), it makes Set 3 active. So subsequent numbers symbols in Set 3 can be encoded without the switch symbol, as in 111000 for 3, 111001 for 4 and so on.
- To return to Set 1 in this case, the code 0000 is used.
- However, when other symbols in Set 3 are encoded from Set 1, Set 3 is not made active.

6.4 Sticky sets

- When switching to Set 3 for encoding numbers (0-9), it becomes active and is said to be sticky till Set 1 is made active using the symbol 0000.
- Encoding Upper case symbols become sticky when switching using 0000 0000.
- Encoding Unicode symbols become sticky when switching using 0000 010, as seen in a subsequent section.
- However, no other set is sticky. Set 1 is default. Set 3 automatically becomes sticky when any numeral is encoded and Upper case letters can be made sticky by using 00000000.
- Symbols in Set 2 are never sticky. Once encoded the previous sticky set becomes active.

6.5 Special symbols

- term in Set 3 indicates termination of encoding. This is used if length of the encoded string is not available. In case the length of encoded string is available, term symbol need not be encoded and encoding can stop with the last symbol encoded. However, the first part of the term symbol needs to be encoded in the last byte after the bits for the last symbol. Further if Unicode set is sticky and active, first it needs to be exited using the exit sequence 11111 00 and then the term symbol should be encoded.

- rpt in Set 3 indicates that the symbol last encoded is to be repeated specified number of times.
- CRLF in Set 2 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 2. Also, in the rare case that only CR appears, another code is provided in Set 2.

6.6 Repeating letters

- If any letter repeats more than 3 times, we use a special code (rpt) shown in Set 3 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.

6.7 Repeating sections

- If a section repeats, the switch code (00) and another horizontal code (110) is used followed by two fields as described next.
- The first field indicates the length of the section that repeats.
- The second field indicates the distance of the repeating section. The distance is counted from the current position.
- The optional third field is coded only if an array of text is encoded. It is a number indicating the set that the section belongs to. If only one set of text is encoded, then this field is not included.
- The first, second and third fields are encoded as explained in the following section "Encoding counts".

6.8 Encoding Counts

- For encoding counts such as length and distance, six codes are used: 0, 10, 110, 1110, 1111, 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, 4 bits would follow, that is, count is between 4 and 19.
- If code is 110, 7 bits would follow, that is, count is between 20 and 147.
- If code is 1110, 11 bits would follow, that is, count is between 148 and 2195.

- If code is 1111, 16 bits would follow, that is, count is between 2196 and 67732.
- This is shown in tabular form below

Code	Range	Number of bits
0	0 to 3	2
10	4 to 19	5
110	20 to 147	7
1110	148 to 2195	11
1111	2196 to 70307	16

6.9 Encoding Unicode characters

- The switch code 00 followed by 111 is used as prefix to indicate that a Unicode character is being encoded.
- First, the unicode number is decoded from the input source depending on how it was encoded, such as UTF-8 or UTF-16 or Wide Character set.
- For the first unicode character, the number decoded is re-coded as a number as shown in section Encoding Counts to the output as it is.
- For subsequent unicode characters, only the difference between the previous character is re-coded to the output. Thus, here, delta coding is used.
- After the code 00 111, another set of prefix-free codes are used, according to the following table, depending on the size (in bits) of the difference.

Code	Range	Number of bits
0	0 to 63	6
10	64 to 4159	12
110	4160 to 20543	14
1110	20544 to 86079	16
11110	86080 to 2183231	21
11111	Special code	-

- The Special code is explained in the next section.
- After 00 111, one of the above codes is used, followed by the sign bit. The sign bit is a single bit. 1 indicates that the number following is negative and 0 indicates that the number following is positive.
- After the sign bit, the unicode value (or difference) is encoded as a number. The number of bits used depends on its size, as shown in the above table.
- After encoding the unicode number, the state returns to Set 1, or whichever set was active earlier, unless continuous unicode encoding was started. This is explained in the next section.

6.10 Encoding continuous Unicode characters

- Since the prefix 00 110 may become an overhead when several Unicode are to be encoded contigously, a continuous unicode encoding code is used (0000 010).
- After 0000 010 is encoded, unicode characters are encoded continously using delta encoding, until a non-unicode character is encountered. When this happens, state is returned to Set 1 using the Special code 11111 00 in the table shown in previous section is used.
- The Special codes are used only when Unicode characters are coded continuously, to indicate special characters and situations occuring in-between. What follows the Special code 11111 is indicated using the table below:

Code	**Character/Situation**
0	Space character
10	Switch
110	Comma (,)
1110	Full stop (.)
1111	Line feed (LF)

- It is found that the above characters appear frequently in between continous Unicode characters and so Special codes are needed to avoid switching back and forth from Set 2.
- Other symbols in Set 2 or Set 3 can also be encoded within continuous Delta encoding mode using the Switch Code in the above table.

6.11 Multi way access for Set 2

- Set 2 can be accessed regardless of which set is active, such as Set 1, Set 3, Continuous delta coding or even when continuous Upper case is active. This is because the symbols occur commonly in both Set 1 and 3 and Unicode symbol sequences.
- For the same reason, the space symbol appears both in Set 1 and Set 3.

6.12 Encoding punctuations

- Some languages, such as Japanese and Chinese use their own punctuation characters. For example full-stop is indicated using U+3002 which is represented visually as a small circle.
- Encoding such special full-stops were supported in Unishox 1 for better compression. However since this was leading to confusion and ambiguity, any special treatment for such punctuations are excluded in Unishox2 and this is left to delta coding. It also does not make much difference in compression ratio.

6.13 Common templates

- Some special templates are known to occur frequently and are encoded using 00 10 00 followed the codes mentioned in the table below.

Code	Situation
0	Template for date, time and phone numbers
10	Hex nibbles lower case
110	Hex GUID lower case
1110	Hex nibbles upper case
11110	Hex GUID upper case
11111	Binary (ASCII 0-31, 128-255)

- The code 0 indicates that one of the codes for Date, Time or Phone number follows, which is encoded according to the following table:

Code	Description	Template
0	Standard ISO timestamp	tfff-of-tfTtf:rf:rf.fffZ
10	Date only	tfff-of-tf
110	US Phone number	(fff) fff-ffff
1110	Time only	tf:rf:rf
1111	Reserved	

Partial matches of the template can also be encoded using this. For example, the string "2021-07-15T20:00:00" can be compressed using above template by specifying how many characters of the template are unused the end. In this case 5 characters are unused.

The encoding sequence would be: 00 10 00 0 <template code> <number of unused letters> <filled template>. The method described in "Encoding counts" section is used to encode <number of unused letters>.

In the template, following are the codes used and the size occupied in bits. Since fewer bits are sufficient to represent a number, it results in lot of savings.

Letter	Bits	Range
o	1	0 to 1
t	2	0 to 3
r	3	0 to 7
f	4	0 to F

The ISO timestamp which is 24 bytes in length compresses to only 9 bytes.

For example, "2021-07-15T16:37:35" would be encoded as 00 10 00 0 0 10 0001 10 0000 0010 0001 0 0111 01 0101 01 0110 011 0111 011 0101. The codes are explained in the table below:

Code	Description
00 10 00	Code for common templates
0	Code for string template
0	Template used (tfff-of-tfTtf:rf:rf.fffZ)
10 0001	Encode count 5 unused at the end
10 0000 0010 0001	2021
0 0111	07
01 0101	15
01 0110	16
011 0111	37
011 0101	35

- The codes 10 and 1110 are used to encode a sequence of lower and upper Hex nibbles respectively. 10 or 1110 is followed by the count of nibbles encoded as explained in the "Encoding counts" section. After this, each nibble is encoded using 4 bits each.
- The code 110 and 11110 are used to encode lower and upper GUIDs respectively. 110 or 11110 is followed by each nibble of the GUID excluding the hyphens.
- Finally the code 11111 is used for encoding binary symbols ranging from ASCII 0 to 31 and ASCII 128 to 255. The prefix code 00 10 00 11111 is used, followed by the number of such binary symbols encoded as explained in "Encoding counts" section. After this each byte is encoded with 8 bits per character.
- Encoding binary symbols this way is not efficient and is only available to cover the entire character set.
- The implementation actually tries to optimize encoding binary sequences by trying to identify UTF-8 sequences within binary sequences in order to get a better compression ratio.

6.14 Compression of frequently occuring sequences

- Provision for six frequently occuring text sequences is available with Unishox 2.
- Depending on the type of text being encoded following sequences have been identified.

Type of text	Frequently occuring sequences
Default (favours all types)	[“: “], [“:], [</], [=“], [“:“], [://]
English sentences	[the], [and], [tion], [with], [ing], [ment]
URL	[https://], [www.], [.com], [http://], [.org], [.net]
JSON	[“: “], [“:], [“,], [}}}], [“:“], [}}]
HTML	[</], [=“], [div], [href], [class], [<p>]
XML	[</], [=“], [">], [<?xml version=“1.0”], [xmlns:], [://]

6.15 Redefinition of Horizontal codes and Presets

- The horizontal codes can be redefined to get better compression ratio, depending on composition of the text to be encoded.
- Several "preset" codes have been identified for achieving better compression ratios for different compositions as below (Codes are for Alpha, Sym, Num, Dict, Delta):
- For preset 1 (Alpha only) there are no horizontal code required. For encoding upper case symbols, just the switch code followed by the letter code is sufficient. Further continuous upper case can be accomplished by using two switch codes.
- The codes marked x in the table are the sets that are not expected in the text.

Preset	**Codes**	**Frequent Sequences**
0 Default (favours all types)	00, 01, 10, 110, 111	Default
1 Alpha only	None *	English sentences
2 Alpha & Numeric only	0, x, 1, x, x	English sentences
3 Alpha, Num & Sym only	0, 10, 11, x, x	Default
4 Alpha, Num & Sym only (Text)	0, 10, 11, x, x	English sentences
5 Favor Alpha	0, 100, 101, 110, 111	English sentences
6 Favor Dictionary	00, 01, 110, 10, 111	Default
7 Favor Symbols	100, 0, 101, 110, 111	Default
8 Favor Umlaut	100, 101, 110, 111, 0	Default
9 No Dictionary	00, 01, 10, x, 11	Default
10 No Unicode	00, 01, 10, 11, x	Default
11 No Unicode (Text)	00, 01, 10, 11, x	English sentences
12 Favor URL	00, 01, 10, 110, 111	URL
13 Favor JSON	00, 01, 10, 110, 111	JSON
14 Favor JSON No Unicode	00, 01, 10, 11, x	JSON
15 Favor XML	00, 01, 10, 110, 111	XML
16 Favor HTML	00, 01, 10, 110, 111	HTML

However, the default horizontal codes work fine for most cases.

7 Implementation

According to the above Rules and Frequency table, a reference implementation has been developed and made available at https://github.com/siara-cc/Unishox as unishox2.c. This is released under Apache License 2.0.

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

- Unishox Compression Library for Arduino Progmem
  https://github.com/siara-cc/Unishox_Arduino_Progmem_lib

- Sqlite3 User Defined Function for Unishox as loadable extension https://github.com/siara-cc/Unishox 2_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

8 Applications

- Compression for low memory devices such as Arduino and ESP8266
- Sending messages over Websockets
- Compression of Chat application text exchange including Emojis
- Storing compressed text in databases
- Faster retrieval speed when used as join keys
- Bandwidth cost saving for messages transferred to and from Cloud infrastructure
- Storage cost reduction for Cloud databases

9 Performance Comparison

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

String	**Length**	**Smaz**	**shoco**	**Unishox**
Hello World	11	10	8	8
The quick brown fox jumps over the lazy dog	43	30	34	30
I would have NEVER said that	28	20	20	19
In (1970-89), $25.9 billion; OPEC bilateral aid [1979-89], $213 million	67	65	52	50

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: 2,988,577 bytes

After Compression using shoco original model: 2,385,934 bytes

After Compression using shoco trained using world95.txt: 2,088,141 bytes
After Compression using Unishox (1024 block size): 1,689,289 bytes
After Compression using Unishox (65536 block size): 1,128,302 bytes

The quote by Kahlil Gibran "Beauty is not in the face. Beauty is a light in the heart." was translated to several languages using Google Translate and the following table shows the savings in bytes for each language. Although some attempt was made by reverse translation to check accuracy, all translations may not be considered correct by natives.

More examples can be found in the presentation https://github.com/siara-cc/Unishox/blob/master/demo/Banner2.ppt?raw=true

Language	Quote by Khalil Gibran	Before	After	Savings
English	Beauty is not in the face. Beauty is a light in the heart.	58	29	50 %
Spanish	La belleza no está en la cara. La belleza es una luz en el corazón.	69	38	45 %
French	La beauté est pas dans le visage. La beauté est la lumière dans le coeur.	76	39	49 %
Portugese	A beleza não está na cara. A beleza é a luz no coração.	60	36	40 %
Dutch	Schoonheid is niet in het gezicht. Schoonheid is een licht in het hart.	71	35	51 %
German	Schönheit ist nicht im Gesicht. Schönheit ist ein Licht im Herzen.	68	35	49 %
Italian	La bellezza non è in faccia. La bellezza è la luce nel cuore.	63	35	44 %
Swedish	Skönhet är inte i ansiktet. Skönhet är ett ljus i hjärtat.	63	34	46 %
Romanian	Frumuseea nu este în fa. Frumuseea este o lumin în inim.	70	44	37 %
Turkish	Güzellik yüzünde deil. Güzellik, kalbin içindeki bir ktr.	72	48	33 %
Polish	Pikno nie jest na twarzy. Pikno jest wiatem w sercu.	60	38	37 %
Afrikaans	Skoonheid is nie in die gesig nie. Skoonheid is 'n lig in die hart.	67	33	51 %
Swahili	Beauty si katika uso. Uzuri ni nuru moyoni.	43	29	33 %
Zulu	Ubuhle abukho ebusweni. Ubuhle bungukukhanya enhliziyweni.	58	39	33 %
Somali	Beauty ma aha in wajiga. Beauty waa iftiin ah ee wadnaha.	57	34	40 %
Azerbaijani	Gözllik üzd deyil. Gözllik qlbd bir iqdr.	60	44	27 %
Uzbek	Go'zallik yuzida emas. Go'zallik - qalbdagi nur.	48	31	35 %
Kurdish	Bedewî ne di rû de ye. Bedewî di dil de ronahiyek e.	55	32	42 %
Malay	Kecantikan bukan di muka. Kecantikan adalah cahaya di dalam hati.	65	38	42 %

As for memory requirements, shoco requires over 2k bytes, smaz requires over 1k. But Unishox requires only around 300 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.

For Unicode text, the codes in section "Encoding Unicode characters" have been selected in such a way that the prefix-code overhead is offset by delta coding and the fact that UTF-8 encoding has more redundant code overhead.

11 Conclusion

As can be seen from the performance numbers, Unishox 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 Unishox 2 by making it available in more languages than just C, such as Javascript, Python, Java, C#.Net.

13 About the Author

Arundale Ramanathan 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.

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):337343, May 1977.

[5] Wikipedia, *Delta Encoding*, https://en.wikipedia.org/wiki/Delta_encoding, updated July 2019.

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

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

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

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

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

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

[12] Arundale Ramanathan, *Unishox - Guaranteed Compression of Short Unicode Strings using Entropy, Dictionary and Delta encoding techniques*, https://github.com/siara-cc/Unishox/blob/master/Unishox_Article_1.pdf?raw=true, August 2019.

