The Journal of Open Source Software

Unishox: A hybrid encoder for Short Unicode Strings

Arundale Ramanathan[1]

1 Independent Researcher

DOI: 10.21105/joss.03919

Software

- Review
- Repository
- Archive

Editor: George K. Thiruvathukal

Reviewers:

- @gradvohl
- @linuxscout

Submitted: 04 November 2021
Published: 18 January 2022

Summary

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

This article discusses a hybrid encoding method for compressing Short Unicode Strings of arbitrary lengths including Latin/English text and printable special characters. This has not been sufficiently addressed by lossless short text encoding methods so far as they have one or more of the following drawbacks:

- They apply only to either English text or Unicode strings and not both
- They have specific criteria based on which best compression can be achieved
- They are not suitable for low RAM devices

To the extent we know, the existing methods available for short string compression are Smaz (Sanfilippo, 2012), Shoco (Schramm, 2015), SCSU, (*A Standard Compression Scheme for Unicode - UTR #6*, 2005), BOCU (Scherer & Davis, 2002), SSE (Juncai Xu et. al, 2017) and AIMCS (Abedi & Pourkiani, 2020).

Statement of need

Space occupied by short strings become significant in memory constrained environments such as Arduino Uno and ESP8266. Text exchange in Chat applications and social media posts is another area where cost savings could be seen using such compression. It is also possible to achieve savings in bandwidth and storage cost by storing and retrieving independent strings in Cloud databases.

Existing Techniques

In information theory, `entropy encoding` is a lossless data compression scheme that is independent of the specific characteristics of the medium (MacKay, 2003).

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 (Shannon, 1948). Therefore, the most common symbols use the shortest codes.

The most popular method for forming optimal prefix-free discrete codes is Huffman coding (Huffman, 1952).

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 (Ziv & Lempel, 1977).

`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 differnce is determined by using the set minus operator or subtraction of values (*Delta Encoding*, 2019).

In contrast to these encoding methods, there are various other approaches to lossless coding including Run Length Encoding (RLE) and Burrows-Wheeler coding (Burrows & Wheeler, 1994).

While programs such as GZip, Deflate, Zip, LZMA and BZ2 that use such technologies are available for general purpose compression, they do not provide optimal compression for short strings. Even though these methods compress far more than what is proposed in this article, they often expand the original source for short strings because the symbol-code mapping also needs to be attached to aid decompression.

Short string compression techniques

Techniques available For compressing short English / Latin text are Smaz and shoco, but are not developed with Unicode in mind.

Smaz is a simple compression library suitable for compressing very short strings (Sanfilippo, 2012). It was developed by Salvatore Sanfilippo and is released under the BSD license.

Shoco is a C library to compress short strings (Schramm, 2015). 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 (Schramm, 2015).

In addition to providing a default frequency table as model, shoco provides an option to re-define the frequency table based on training text (Schramm, 2015).

For compressing Unicode sequences, three technologies are available: SCSU (*A Standard Compression Scheme for Unicode - UTR #6*, 2005), BOCU (Scherer & Davis, 2002), and FAST (Pavel Studený, 2008). Ewell (2004) published a survey of these compression algorithms.

The Standard Compression Scheme for Unicode (SCSU) is defined in Unicode Technical Standard #6 and is based on a technique originally developed by Reuters. The basic premise of SCSU is to define dynamically positioned windows into the Unicode code space, so that characters belonging to small scripts (such as the Greek alphabet or Indic abugidas) can be encoded in a single byte, representing an index into the active window. These windows are preset to blocks expected to be in common use (e.g., Cyrillic), so the encoder doesn't have to define them in these cases. There are also static windows that cannot be adjusted (*A Standard Compression Scheme for Unicode - UTR #6*, 2005; Ewell, 2004).

The Binary Ordered Compression for Unicode (BOCU) concept was developed in 2001 by Mark Davis and Markus Scherer for the ICU project. The main premise of BOCU-1 is to encode each Unicode character as the difference from the previous character, and to represent small

differences in fewer bytes than large differences. By encoding differences, BOCU-1 achieves the same compression for all small alphabetic scripts, regardless of the block in which they reside (Ewell, 2004; Scherer & Davis, 2002).

It is to be noted that SCSU is a Unicode Technical Standard (UTS#6) and BOCU is published as a Unicode Technical Note (UTN#6), although both have the same number assigned (6).

Fast Compression Algorithm For Unicode Text (FAST) is a compression algorithm developed based on the Lempel Ziv algorithm (Ziv & Lempel, 1977). Essentially it achieves faster compression by finding repeating unicode sequences instead of repeating bytes. There are other assumptions and variations made to LZ technique in addition to this (Pavel Studený, 2008).

AIMCS is an Artificial Intelligence based Method for Compression of Short Strings, which is specifically designed for compression of strings with the size of less than 160 characters (tiny strings) (Abedi & Pourkiani, 2020).

SSE is a technique where texts are pre-processed by a method named `sort and set empty` and are then compressed through the traditional lossless compression methods (Juncai Xu et. al, 2017).

Further four different methods available for compressing short messages have been discussed in this paper (Gardner-Stephen et. al, 2013) including Smaz and other closed source techniques.

This research

A hybrid encoding method is proposed relying on the three encoding techniques `viz.` Entropy encoding, Dictionary coding and Delta encoding methods for optimal compression.

While existing techniques focus on either Unicode character sequences or only English characters, Unishox uses multiple techhniques to achieve the best compression ratio all round.

For Unicode, Delta encoding is proposed because usually the difference between subsequent symbols is quite less while encoding text of a particular language. SCSU is slightly better with switching windows, but overall it was found that plain Delta coding works well considering that usually there is only one language text to be compressed and some languages span a lot of windows.

SCSU and BOCU do have special features for Unicode that Unishox does not address such as dynamic windows, binary order maintenance, XML suitability and MIME friendliness. Unishox uses plain delta encoding to achieve the best compression.

For English letters, unlike shoco, a fixed frequency table is proposed, generated based on the characterestics of English language letter frequency. The research carried out by Oxford University (*What is the frequency of the letters of the alphabet in English?*, 2012) and other sources (*Statistical Distributions of English Text, Archived from the Original*, 2017) have been used to arrive at a unique method that takes advantage of the conventions of the language.

A single fixed model is used 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. It can be seen that this performs better than pre-trained model of shoco (See performance section).

This model, described in the subsequent section, along with a set of rules for switching between the pre-defined sets of symbols in the model are used for encoding and decoding text.

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, two sets of prefix-free codes are used.

The first set consists of 28 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. These are called vertical codes (vcodes).

The second set consists of 5 codes, which by default will be 00, 01, 10, 110, 111. These are called 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, several sets of letters are formed as shown in the table below and some rules are formed 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	<length>	<code>
010	sp	{	,	<distance>	<sign>
011	e / E	}	.		<delta>
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		
1110110	h / H	’	7		
1110111	u / U	tab	8		
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		

Rules

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.

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 00 00 011.
- If uppercase letters appear continuously, then the encoder may decide to switch to upper case using the prefix 00 00 00 00. After that, the same codes for lower case are used to indicate upper case letters until the code sequence 00 00 is used again to return to lower case.

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.

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.

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.

Repeating letters

- If any letter repeats more than 3 times, a special code (rpt) is used as 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.

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 index of the array that the section belongs. If only one 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.

Encoding Counts

- For encoding counts such as length and distance, five 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 67732	16

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.
- For the first unicode character, the number decoded is re-coded to the output as it is, using 00 111 followed by a code as shown in the table below followed by a sign bit 0 (positive), followed by given number of bits shown in the table, depending on the range that the code belongs.

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.
- For subsequent unicode characters, only the difference between the previous character is re-coded to the output, using sign bit as 1 if the difference is negative. Thus, here, delta coding is used.
- 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 the range, 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.

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 an English 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 (,)

Code	Character/Situation
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.

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.

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 the earlier version of Unishox for better compression. However since this was leading to confusion and ambiguity, any special treatment for such punctuations are excluded in the present version of Unishox (2) and this is left to delta coding. It also does not make much difference in compression ratio.

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

Code	Description	Template
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

- Using this method, 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.

Compression of frequently occuring sequences

- Provision for six frequently occuring text sequences is available with Unishox.
- 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:], [://]

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. Termination of encoding is accomplished by encoding 3 or 4 switch codes continuously depending on whether continuous upper case encoding is active or not.
- 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

Preset	Codes	Frequent Sequences
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.

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
- Some people even use it for obfuscation

Implementation

According to the above Rules and Frequency table, an implementation has been developed licensed under Apache License 2.0.

Unishox has been hosted on Github and used in several open source projects shown below:

- Unishox
  https://github.com/siara-cc/Unishox
- Unishox for Javascript
  https://github.com/siara-cc/Unishox/_JS
- Python bindings for Unishox
  https://github.com/tweedge/unishox2-py3
- Unishox 1 ported to Python for Tasmota
  https://github.com/arendst/Tasmota/tree/development/tools/unishox
- 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/_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

Performance Comparison

The performance of Unishox was compared with the various implementations already available for short strings and shown in subsequent sections.

Comparison with Unicode compression techniques

Language and Text	Size	Unishox	SCSU	BOCU
English	58	30	58	58
Chinese	49	36	36	37
Spanish	69	38	67	71
Hindi	144	53	55	55
Bengali	117	41	48	47
Portugese	60	36	55	63
Russian	82	44	48	53
Japanese	61	39	37	45
Punjabi	141	51	57	59
Marathi	142	52	55	58
Telugu	104	39	42	44
Turkish	72	49	64	78
Korean	82	45	61	60
French	76	39	73	79
German	68	36	66	70
Vietnamese	82	59	72	83
Tamil	128	49	50	52

- All sizes are in bytes.

The above table compares the compression performance between SCSU, BOCU and Unishox for languages that are spoken by over 75 million people (according to Wikipedia).

The text used is translation of Kahlil Gibran's quote "Beauty is not in the face. Beauty is a light in the heart." in the above languages. The actual translated text could not be displayed due to limitation of Markdown format.

Disclaimer: Natives may not consider all translations to be accurate as they were translated online, although some attempt was made to check accuracy by reverse translation.

Comparison with non-Unicode compression techniques

String	Size	Unishox	Smaz	Shoco
Beauty is not in the face. Beauty is a light in the heart.	58	30	31	46
The quick brown fox jumps over the lazy dog.	44	31	31	38

String	Size	Unishox	Smaz	Shoco
WRITING ENTIRELY IN BLOCK CAPITALS IS SHOUTING, and it's rude	63	49	72	63
Grawlix is a string of typographical symbols (such as %@$|*!) coined in the 1960s	82	60	58	63
Rose is a rose is a rose is a rose.	35	12	20	25
Gravitational Constant (G): 6.67300 x 10^{}-11 m^{}3 kg^{}-1 s^{}-2	59	50	65	51
039f7094-83e4-4d7f-aa38-8844c67bd82d	36	18	53	36
2021-07-15T16:37:35.897Z	24	9	32	24
(760) 756-7568	14	7	20	14
This is a looooooooooooooooooooong string	42	15	32	25

- All sizes are in bytes.

The above table compares the compression performance of Smaz, shoco and Unishox for different types of strings.

Comparison of file compression

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

Memory requirements

As for operating memory required, 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.

Speed

Unishox was found to be the slowest of all since employs several to achieve the best compression. However this should not be too much of an issue in most cases when a single string or few strings are handled at a time.

Conclusion

As can be seen from the performance numbers, Unishox performs better than available techniques. It can also be seen that it provides optimal compression for text, numbers and special characters in different languages all round.

It is especially useful in memory constrained environments such as embedded devices and sending text messages over websockets to implement Chat bots and applications.

Further work

It is proposed to achieve better compression by choosing better codes during the course of compression using a self-learning process.

It is also proposed to make Unishox available in more languages than just C, Javascript and Python, such as Java and C#.Net. It is also proposed to make it available for more platforms.

Acknowledgements

The author is sincerely thankful to the following people who have notably contributed towards the development of Unishox implementation:

- Thanks to Jonathan Greenblatt for his port of Unishox2 that works on Particle Photon
- Thanks to Chris Partridge for his port of Unishox2 to CPython and his comprehensive tests using Hypothesis and extensive performance tests.
- Thanks to Stephan Hadinger for his port of Unishox1 to Python for Tasmota
- Thanks to Luis Díaz Más for his PRs to support MSVC and CMake setup
- Thanks to James Z.M. Gao for his PRs on improving presets, safety checks, terminator codes, unit tests, bug fixes, documentation and more

References

A standard compression scheme for unicode - UTR #6. (2005). Unicode Consortium. https://www.unicode.org/reports/tr6/tr6-4.html

Abedi, M., & Pourkiani, M. (2020). AIMCS: An artificial intelligence based method for compression of short strings. *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 311–318. https://doi.org/10.1109/SAMI48414.2020.9108719

Burrows, M., & Wheeler, D. (1994). A Block-Sorting Lossless Data Compression Algorithm. In *Research Report 124, Digital Equipment Corporation, Palo Alto, CA, USA*. https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf

Delta encoding. (2019). Wikipedia. https://en.wikipedia.org/wiki/Delta_encoding

Ewell, D. (2004). *A survey of Unicode compression, Unicode Technical Note #14, version 1*. http://www.unicode.org/notes/tn14/

Gardner-Stephen et. al. (2013). Improving compression of short messages. *International Journal of Communications, Network and System Sciences*, *6*, 497. https://doi.org/10.4236/ijcns.2013.612053

Huffman, D. (1952). A method for the construction of minimum-redundancy codes. *Proc. IRE*, *40*(9), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898

Juncai Xu et. al. (2017). SSE lossless compression method for the text of the insignificance of the lines order. *ArXiv*, *abs/1709.04035*. https://arxiv.org/pdf/1709.04035

MacKay, D. J. C. (2003). *Information theory, inference and learning algorithms*. Cambridge University Press. https://doi.org/10.1109/tit.2004.834752

Pavel Studený, O. S. A., Ondřej Holeček. (2008). *Fast Compression Algorithm For Unicode Text, Unicode Technical Note #31, version 2*. http://www.unicode.org/notes/tn31/

Sanfilippo, S. (2012). *SMAZ - compression for very small strings*. github.com. https://github.com/antirez/smaz

Scherer, M. W., & Davis, Mark. (2002). *BOCU-1: MIME-Compatible Unicode Compression, Unicode Technical Note #6, version 1*. http://www.unicode.org/notes/tn6/

Schramm, C. (2015). *Shoco: A fast compressor for short strings*. https://ed-von-schleck.github.io/shoco

Shannon, C. E. (1948). A mathematical theory of communication. *Bell Syst. Tech. J.*, *27*(3), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Statistical distributions of english text, archived from the original. (2017). data-compression.com. https://web.archive.org/web/20170918020907/http://www.data-compression.com/english.html

What is the frequency of the letters of the alphabet in English? (2012). Oxford University Press, Oxford Dictionary.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, *23*(3), 337–343. https://doi.org/10.1109/TIT.1977.1055714