# Shox96 - Guaranteed Compression of Printable Short Strings

Arundale R, Charumathi A, Harsha N

February 8, 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 because the symbol-code mapping also needs to be sent along with the compressed bit stream. Eventhough these methods compress far more than what we are proposing, these methods often expand the original source 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 5 codes, which are 0, 10, 110, 1110, 1111.

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.

| **code →** | **10** | **0** | **110** | **1110** | | **1111** | |
|---|---|---|---|---|---|---|---|
| **↓ code** | **Set 1** | **Set 1a** | **Set 1b** | **Set 2** | **Set 2a** | **Set 3** | **Set 3a** |
| **00** | switch | l / L | term | switch | . | switch | @ |
| **010** | sp / tb | c / C | f / F | 9 | - | ; | ? |
| **011** | e / E | cr+lf | y / Y | 0 | , | : | ’ |
| **100** | t / T | d / D | v / V | 1 | / | < | ˆ |
| **1010** | a / A | h / H | k / K | 2 | = | > | # |
| **1011** | o / O | u / U | q / Q | 3 | + | * | _ |
| **1100** | i / I | p / P | j / J | 4 | sp | “ | ! |
| **1101** | n / N | m / M | x / X | 5 | ( | { | \ |
| **1110** | s / S | b / B | z / Z | 6 | ) | } | \| |
| **11110** | r / R | g / G | rpt | 7 | $ | [ | ˜ |
| **11111** | Set2a / & | w / W | dict | 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 l is encoded as 00000, c 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 f is encoded as 00110010, y as 00110011 and so on. Note that the terminator symbol is encoded as 0011000.

 

### 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 and &.
- 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 H is encoded as 00100001010.
- Similarly, for uppercase letters in Set 1b, the prefix 001000110 is used. So the symbol K 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 1110. So the symbol 9 is encoded as 001110010.
- 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 Set2a, if Set 2 is active, the prefix 00 is used followed by 0 and the corresponding code for the symbol. For example, if Set 2 is active, + is encoded as 0001011.
- If Set 1 is active, the prefix 11111 is used to encode symbols in Set 2a. So for +, the code would be 111111011 in this case.

### 6.4 Other symbols (Set 3)

- The special characters in Set 3 can be encoded by using the prefix 001111 followed by the corresponding code for the letter. Example: ; is encoded as 001111010.
- The symbols in Set 3a are encoded using the prefix 00111100. Example: ? is encoded as 00111100010.

 

### 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 001110 and Upper case letters can be made sticky by using 00100010.
- Symbols in Set 3 and Set 3a 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. As of now this is not implemented yet.
- 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. This is also not implemented yet.
- rpt and dict would be eventually implemented. These do not occur in Short strings frequently. However, these are expected to have an impact and releatively longer strings.
- CRLF in Set 1a is encoded as a single code. It will be expanded as CRLF or LF depending on the target OS. In this case, the term lossless encoding is slightly deviated.

### 6.7 Dual access for Set 2a

- Set 2a 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 2a.

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

 

## 7 Frequency table

| ASCII Code | Letter | Code | Length |
|---|---|---|---|
| 32 | | 010 | 3 |
| 33 | ! | 001111001100 | 12 |
| 34 | “ | 0011111100 | 10 |
| 35 | # | 001111001010 | 12 |
| 36 | $ | 1111111110 | 10 |
| 37 | % | 1111111111 | 10 |
| 38 | & | 001011111 | 9 |
| 39 | ’ | 00111100011 | 11 |
| 40 | ( | 111111101 | 9 |
| 41 | ) | 111111110 | 9 |
| 42 | * | 0011111011 | 10 |
| 43 | + | 111111011 | 9 |
| 44 | , | 11111011 | 8 |
| 45 | - | 11111010 | 8 |
| 46 | . | 1111100 | 7 |
| 47 | / | 11111100 | 8 |
| 48 | 0 | 001110011 | 9 |
| 49 | 1 | 001110100 | 9 |
| 50 | 2 | 0011101010 | 10 |
| 51 | 3 | 0011101011 | 10 |
| 52 | 4 | 0011101100 | 10 |
| 53 | 5 | 0011101101 | 10 |
| 54 | 6 | 0011101110 | 10 |
| 55 | 7 | 00111011110 | 11 |
| 56 | 8 | 00111011111 | 11 |
| 57 | 9 | 001110010 | 9 |
| 58 | : | 001111011 | 9 |
| 59 | ; | 001111010 | 9 |
| 60 | < | 001111100 | 9 |
| 61 | = | 111111010 | 9 |
| 62 | > | 0011111010 | 10 |
| 63 | ? | 00111100010 | 11 |
| 64 | @ | 0011110000 | 10 |

| **ASCII Code** | **Letter** | **Code** | **Length** |
|---|---|---|---|
| 65 | A | 00101010 | 8 |
| 66 | B | 00100001110 | 11 |
| 67 | C | 0010000010 | 10 |
| 68 | D | 0010000100 | 10 |
| 69 | E | 0010011 | 7 |
| 70 | F | 001000110010 | 12 |
| 71 | G | 001000011110 | 12 |
| 72 | H | 00100001010 | 11 |
| 73 | I | 00101100 | 8 |
| 74 | J | 0010001101100 | 13 |
| 75 | K | 0010001101010 | 13 |
| 76 | L | 001000000 | 9 |
| 77 | M | 00100001101 | 11 |
| 78 | N | 00101101 | 8 |
| 79 | O | 00101011 | 8 |
| 80 | P | 00100001100 | 11 |
| 81 | Q | 0010001101011 | 13 |
| 82 | R | 001011110 | 9 |
| 83 | S | 00101110 | 8 |
| 84 | T | 0010100 | 7 |
| 85 | U | 00100001011 | 11 |
| 86 | V | 001000110100 | 12 |
| 87 | W | 001000011111 | 12 |
| 88 | X | 0010001101101 | 13 |
| 89 | Y | 001000110011 | 12 |
| 90 | Z | 0010001101110 | 13 |
| 91 | [ | 0011111110 | 10 |
| 92 | \ | 001111001101 | 12 |
| 93 | ] | 0011111111 | 10 |
| 94 | ^ | 00111100100 | 11 |
| 95 | _ | 001111001011 | 12 |
| 96 | ` | 0011110011111 | 13 |

 

| ASCII Code | Letter | Code | Length |
|---|---|---|---|
| 97 | a | 1010 | 4 |
| 98 | b | 0001110 | 7 |
| 99 | c | 000010 | 6 |
| 100 | d | 000100 | 6 |
| 101 | e | 011 | 3 |
| 102 | f | 00110010 | 8 |
| 103 | g | 00011110 | 8 |
| 104 | h | 0001010 | 7 |
| 105 | i | 1100 | 4 |
| 106 | j | 001101100 | 9 |
| 107 | k | 001101010 | 9 |
| 108 | l | 00000 | 5 |
| 109 | m | 0001101 | 7 |
| 110 | n | 1101 | 4 |
| 111 | o | 1011 | 4 |
| 112 | p | 0001100 | 7 |
| 113 | q | 001101011 | 9 |
| 114 | r | 11110 | 5 |
| 115 | s | 1110 | 4 |
| 116 | t | 100 | 3 |
| 117 | u | 0001011 | 7 |
| 118 | v | 00110100 | 8 |
| 119 | w | 00011111 | 8 |
| 120 | x | 001101101 | 9 |
| 121 | y | 00110011 | 8 |
| 122 | z | 001101110 | 9 |
| 123 | { | 0011111101 | 10 |
| 124 | — | 001111001110 | 12 |
| 125 | } | 0011111110 | 10 |
| 126 | ˜ | 0011110011110 | 13 |

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. This is released under Apache License 2.0.

 

## 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 | 31 |
| 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 | 48 | 45 |

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: 1904252 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.

Further, we will be implementing the part for character repetition and dictionary support in the near future.

## 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.

 