Duncode Characters Shorter

Changshang Xue

Abstract—
Many encoders are adopted in various texts to transform characters into bytes. Local encoders (ASCII, GB-2312) always encode specific characters into shorter bytes. But only universal encoders (UTF-8, UTF-16) can encode whole Unicode set with more space cost and become more and more having general approval. And some other encoders (SCSU, BOCU-1 and binary encoders) lack of self-synchronizing.
Duncode devotes to encode whole Unicode characters in high space efficiency as local encoders. Multi characters of a string can be compressed into a Duncode unit with less bytes. Duncode shows a great advantage over UTF8 in space efficiency at the expense of less self-synchronizing identification information. The application is released at https://github.com/laohur/duncode.
We also build a benchmark, including 179 languages, for character encoder evaluation on different languages at https://github.com/laohur/wiki2txt.

Index Terms—Duncode, character encoder, UTF-8, Unicode, text compression, text storage

1 Introduction

ENCODING characters to bytes is the first step of processing text with computers. A good character encoder would be including all characters, saving space and robust (including lossless). However, there is no one character encoder takes these features all above. Including all characters and robust are foundational encode features. Smaller size is helpful for saving storage space and network bandwidth [1].

The Unicode (ISO/IEC 10646) [2] coded character set (Universal Character Set, UCS) [3] is the largest of its kind. Including all characters means storing and transmitting Unicode characters in generally speaking today. It is a rigid demand for exchange texts over the world [4]. Further more, a character encoder is better perform well on every language.

Bytes/Characters(Symbol Length) shows how long a symbol are encoded into bytes for an encoded byte sequence in average. We usually use this index to measure the space efficiency for encoders. It may fluctuate for different characters inside one encoder.

$$Bytes/Characters = \frac{Number\,of\,Bytes}{Number\,of\,Characters} \quad (1)$$

Self-synchronizing means whether a code unit starts a character can be determined without examining earlier code units. Letting a reader start anywhere and immediately detect byte sequence boundaries helps an encoder to be robust enough to keep bad character from text away. Lossy is not in our target [5]. Some encoders stick to forbidden lookup false positive on search string directly on bytes.

Local encoder and universal encoder are two main stream encoders for encoding thousands of languages [6]. Local encoders (ASCII [7], ISO8859 [8]) come out earlier with shorter encoded bytes length. But their character sets often only contain specific symbols and can not satisfy the demand on exchange. Some of them lack of the ability of self-synchronizing on decoding. They can be corrupt on exchange, especially on the Internet. They decline now for this reason.

Universal encoders can encode the whole Unicode symbols. Some encoders (Unicode Transformation Format,UTF) [3] achieve this goal. Errors are limited in one unit. They become the most popular encoders on exchange. In general, they work with other binary compressors on the Internet [9]. Some other universal encoders (SCSU [10], BOCU-1 [11]) aim to gain better encode efficiency by inserting tag byte as leading byte for new block in sequence. They succeed in space efficiency with the expense of robust. Thus, they can not decode units inner a byte stream. General compressors [12] usually performs better [13] than character encoders with the aid of general compress algorithms [14] [15] [16]. But they lack of the ability of self-synchronizing on decoding. And binary encoders work for bytes after texts are already encoded.

Most character encoders (local encoders, UTFs) encode each character as an isolated unit. Although the character set of a text may be up to millions in size, neighbor characters are more likely within one same language. These characters can be encoded as a language prefix with some letters of small size alphabet. The Unicode Point of a character cat be decomposed as Alphabet ID and Letter Index in its alphabet. Then the continuous symbols can be compressed by sharing just one Alphabet ID in a Duncode Unit. By sharing common Alphabet ID, Duncode shrinks the size of encoded bytes.

Comparing with UTF-8, Duncode use Tail Byte(Last Byte) to segment Duncode units in bytes stream instead of Leading Byte(First Byte) in UTF-8. A Duncode unit are only 1-4 bytes. It is luxury the encoding unit length into bytes. So the Duncode abandons the unit length message in segment identifier byte to keep the ability of self-synchronizing. For Duncode only keeps the unidirectional compatibility with ASCII, false positive error may occur on search string directly on Duncode bytes; more on that in Section 3 and Section **??**.

In this paper, we introduced a high space efficiency text encoder. Duncoder represents the whole Unicode characters with high space efficiency and self-synchronizing, at the cost of some forbidden lookup false positive. To evaluating compression

• *Beijing, China E-mail: mrxue@pku.edu.cn*

Manuscript received January 2023

performance [17] for character encoders on all languages, we build a tool to collect 179 languages corpus from wikipedia. The result and detail of test in Section 4 and Section **??**.

2 Related Work

2.1 Local Encoder

ASCII [7] is the most famous encoder with a 128 symbols set. ASCII encode most frequent English letters, text symbols and computer orders into one byte. It is usually used in English and rewrite computer orders. Western Europe countries usually fill the last 128 blank position of ASCII byte with their custom characters. Most later encoders keep a compatibility of ASCII. Other languages often need more bytes to encode their characters. For example, GB2312 [18] takes 2 bytes to present thousands of hot Chinese characters. These local encoders usually bind a limit character set and encodes with high space efficiency. However, conversion is a necessary obstacle on exchange and it is not an easy work [19]. Unfortunately, character corruption and wide word injection often occurs in this scenario [20]. In some extreme conditions, whole text corrupt for just only one bad byte.

2.2 Universal Encoder

Character sets become independent from encoders for more universal. With including more than 140,000 characters and over 300 languages today, Unicode/ISO 10646 [2] tries to collect whole characters in the world. Unicode becomes the most popular character set and UTF (Unicode Transformation Format) [21] becomes the most popular text encoders. As the most popular encoders today, UTF-8 [22] and UTF-16 [23] can encode whole Unicode characters. The symbol length of UTF-16 is 2 in general and UTF-16 costs two times space than ASCII for English. The symbol length of UTF-8 varies from 1 to 6. UTF-8 keeps ASCII as original and wraps other characters into longer bytes. It is economic just only for Latin languages.

Some other universal encoders (SCSU [10], BOCU-1 [11]) aim to gain better encode efficiency by inserting tag byte as leading byte for new block in sequence. They can not decode units from a byte stream.

Binary compressor usually performs better than character encoders, especially inspired by general compressing algorithms [24] [25] [26] [27] [28]. But they work for bytes not characters. [29] optimize binary compression algorithms for UTF-8 byte sequence. [30] shows that binary compressor works well on compression of file. [31] and [32] prefer compressing outside of text.

Some works [33] [34] [35] [36] aim to speeding up text search on compressed file.

3 Methodology

Table 1: UTF-8 Unit. The UTF-8 unit of character ”v” costs 1∼6 bytes.

Bytes	Characters	Bytes/Characters	Byte Sequence in Binary					
1	1	1	0vvvvvvv					
2	1	2	110vvvvv	10vvvvvv				
3	1	3	1110vvvv	10vvvvvv	10vvvvvv			
4	1	4	11110vvv	10vvvvvv	10vvvvvv	10vvvvvv		
5	1	5	111110vv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	
6	1	6	1111110v	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv	10vvvvvv

3.1 Tail Byte for Self-Synchronizing

For a byte sequence of UTF-8 in Table 1, only ASCII unit start with byte ’0xxxxxxx’ and all non-ASCII character unit start with byte ”1xxxxxxx”. In a multibyte unit, only the first byte looks like ”11xxxxxx” and others like ”10xxxxxx”. The length of bit ”1” sequence in first byte’s head determines the number of unit bytes. The first byte of UTF-8 unit not only segments the byte sequence but also indicates the length of this unit. Indeed, the unit length just varies from 1 to 6. The UTF-8 unit length flag bits take many bits but carry a few messages. So we give up this message in Duncode.

Duncode code unit conveys less synch information than UTF-8 code unit. Only the **Tail Byte** (last byte of unit) of Duncode unit are encoded as ”0xxxxxxx” and others as ”1xxxxxxx” in Table 2. For ASCII symbols, the only one byte is also the tail byte. It is easy to find unit boundaries from Duncode byte sequence. The unit length varies from 1-4 by decoding deque dynamically. Just only three bytes unit(2^21) is enough to store the whole Unicode symbols. Only first bit is used as a flag foreach Duncode byte.

We get their zones which unit length ranges from 1 to 3 byte in this way. We call them zone ”ascii”, ”byte2” and ”isolate” in 3. Each one of them only hold one character in every unit.

The tail byte may be an ASCII symbol or a part of a longer unit. It may cause lookup false positive problems when search string directly on encoded byte sequence.

Table 2: Duncode Unit. We encode different symbols into various Duncode zones. A single character "x" can be encoded as a 1/2/3 bytes unit in zone "ascii", "byte2" and "isolate". A string of 2/3 symbols can be compressed as a 4-bytes unit for some certain languages. 3 symbols (x, y, z) of alphabet "nnn" are encoded as one unit in zone "bit8" or "bit7".

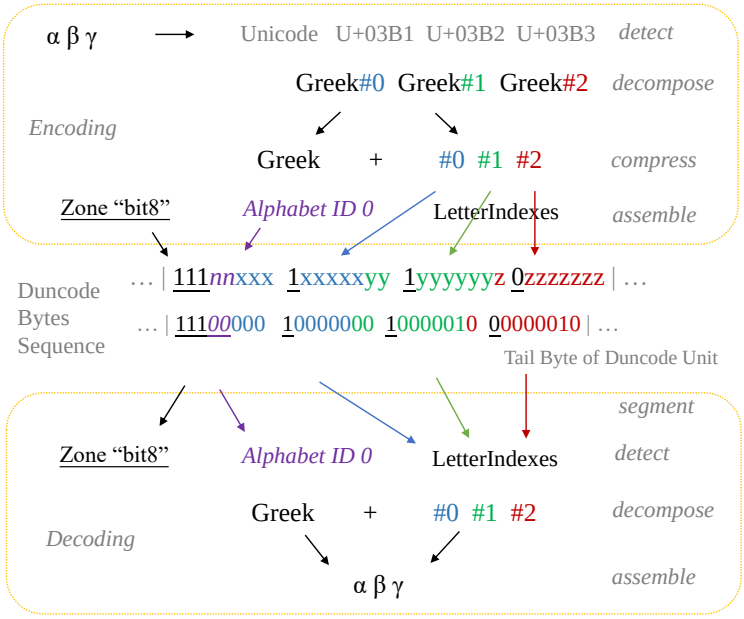
ZoneId	ZoneName	Byte Sequence in Binary				Characters	Languages	Bytes/Characters
0	ascii				0xxxxxxx	x	ascii	1
1	byte2			1xxxxxxx	0xxxxxxx	x	Latin, HanZi,⋯	2
2	isolate		1xxxxxxx	1xxxxxxx	0xxxxxxx	x	rare symbols	3
3	bit8	111nnxxx	1xxxxxyy	1yyyyyyz	0zzzzzzz	x, y, z	Greek⋯	1.33
4	bit7	1nnnnnnn	1xxxxxxx	1yyyyyyy	0zzzzzzz	x, y, z	Devanagari⋯	1.33

3.2 Compress Multi Characters into One Duncode Unit

Almost every self- character encoder maps a character to one encode unit. However, the characters in a sentence belong to one common language in most case. And it always encodes redundant message when encoding these characters as isolated units. We can decompose a character to a **Alphabet ID** and **Letter Index** in the alphabet. This alphabet is a character subset and binds to certain languages. Each of them combines one or two Unicode blocks in common. The index of a Duncode alphabet is named as "Alphabet ID". The offset of this character in this alphabet is called "Letter Index".

The multi characters can be represented as several Letter Indexes with a shared Alphabet ID as a prefix. Their alphabets often contains no more than 128/256 letters. A Letter Index only takes 7/8 bits for each alphabet. Then we can compress 3 symbols into a 4-bytes unit for these languages. Symbol length can reach to 1.33 in zone "bit8" and "bit7" after compression.

Figure 1: Compress ”αβγ” into one Duncode Unit in zone bit8. the string ”αβγ” are decomposed into an Alphabet ID ”Greek” and three Letter Indexes ”0,1,2”. Then they are assembled into a Duncode unit in zone bit8. These units end with Tail Byte and easy to be identified.



Take an example, if we want to encode a Greek string “αβγ” into Duncode, we find these characters are Greek and get Alphabet ID 0 in mapping at first. Then we calculate their Letter Indexes are 0,1,2 in Greek alphabet. So we allocate these Alphabet ID and Letter Indexes to Duncode zone bit8 by Alphabet ID 1. We get a 4 bytes Duncode unit in final.

3.3 Auto Adaptive

If a longer unit is not full of symbols, it will convert to a shorter unit. A same character may locate in different zones. i.e., the string ”α” will be encoded as two bytes unit (zone byte2) instead of four bytes unit (zone bit8).

4 Experiments and Results

4.1 Zones and Alphabets

As Table 3 shows, We keep ASCII as original in zone ascii(lower to differ from ASCII).

Latin, General Punctuation, Hiragana, Katakana and hot Chinese are placed in zone byte2 with a symbol length 2. They are often mixed with other languages or have too great alphabet size to be compressed.

The languages of smaller than 256 symbols in alphabet can be compressed with four bytes unit. We put common languages (<=128 letters) in zone bit7 and some languages (Arabic, Russian..., 128-256 letters) in zone bit8.

Some Unicode Blocks share one Alphabet. For example, Unicode Blocks ”Greek and Coptic” and ”Ancient Greek Numbers” are in one Duncode Alphabet. There are about 300 Unicode Blocks to store all Unicode Symbols. In this way, we union them as about 100+ Duncode Alphabets.

To determine which Alphabet a character belongs to, Duncoder stored about 300 Unicode Blocks ranges as a list and about 2^14 characters as a map for zone byte2.

4.2 Benchmark

To evaluating compression performance on all languages, we collect 179 languages texts as our corpus dataset. All of corpus in this experiment are downloaded from https://dumps.wikipedia.org and extracted into 1-MB-size (in maximum) texts via wiki2txt[1]. We mainly compare Duncoder with UTF-8. Because UTF (Unicode Transformation Format) have all the main attributes as Duncoder and UTF-8 is the most popular and efficient one. The full result shows in Table **??**.

4.3 Results Analysis

The Bytes/Characters(Symbol Length) of Duncoder in these languages meets the expected performance. Almost Duncoder performs better than UTF-8 for every language. We select some typical languages for deep inspection. Their results of encoders are in Table 4.

These encoders show little difference in English and French. Both English and French texts are most of ASCII and Latin characters. All these encoders encode ASCII symbols in one byte and encode Latin symbols in two bytes. As for the dominance of ASCII symbols, their texts are encoded as about one byte per character on average.

1. https://github.com/laohur/wiki2txt

Table 3: Languages of Duncode Zones. Different languages are allocated into different zones. We encode on ASCII symbol as one byte in zone ascii. We encode a popular character into a unit with two bytes in zone byte2. We encode a rare character into a unit with three bytes in zone isolate. About a hundred languages with 128-letters alphabets are encoded in zone bit7. And 4 languages with over 128 letters alphabets are encoded in zone bit8.

Zone	Encode Method	Character Sets	Capacity of Symbol Set
ascii	=ASCII	ASCII	2^7 symbols
byte2	1-1 map	0x0080∼0x07ff, hot HanZi, Tibetan, Mongolian, ...	2^14 symbols
isolate	ZoneOffset + Unicode Point	rare symbols, isolate symbols, symbols of big Alphabet	2^21 symbols
bit8	ZoneOffset + AlphabetOffset + LetterIndexs, then compress 3 symbols into a unit	Greek and Coptic, Cyrillic, Arabic, Myanmar	4 languages of each 256-letters alphabet
bit7	ZoneOffset + AlphabetOffset + LetterIndexs, then compress 3 symbols into a unit	common languages	128 languages(Alphabets) of each 128-letters alphabet

Table 4: Benchmark Duncode on Various Texts.

language	wiki_file_1m	n_chars	n_bytes (utf8)	n_bytes (duncode)	n_bytes/n_chars (utf8)	n_bytes/n_chars (duncoder)	utf8/duncode (size)
English	en.txt	1,054,002	1,066,474	1,061,949	1.01	1.01	100.43%
French	fr.txt	1,054,065	1,096,721	1,094,085	1.04	1.04	100.24%
Arabic	ar.txt	1,103,308	1,855,164	1,462,890	1.68	1.33	126.82%
Russian	ru.txt	1,049,337	1,821,554	1,398,275	1.74	1.33	130.27%
Chinese	zh.txt	1,052,649	2,420,113	1,740,409	2.30	1.65	139.05%
Japanese	ja.txt	1,051,113	2,689,017	1,872,561	2.56	1.78	143.60%
Korean	ko.txt	1,048,759	2,103,649	2,087,029	2.01	1.99	100.80%
Abkhazian	ab.txt	1,049,130	1,846,144	1,411,219	1.76	1.35	130.82%
Burmese	my.txt	1,052,890	2,820,647	1,456,799	2.68	1.38	193.62%
Central Khmer	km.txt	1,081,258	2,890,359	1,516,227	2.67	1.40	190.63%
Tibetan	bo.txt	1,053,038	3,108,029	2,080,294	2.95	1.98	149.40%
Yoruba	yo.txt	1,050,996	1,230,927	1,193,098	1.17	1.14	103.17%

Arabic and Russian texts can be compressed by Duncoder. In this reason, their sizes are smaller than UTF-8 (2 bytes/char → 1.33 bytes/char). Because many blank spaces in these texts, their average byte length per character among 1 to 2 in both Duncoder and UTF-8.

Chinese, Japanese and Korean are of huge character set. As a comparison, we just only optimize Duncoder for Hanzi (Chinese characters, CJKV Unified Ideographs), not Hangul Jamos or Syllables. The popular Hanzi are stored in a map. Then they can be encoded as two bytes. So Chinese and Japanese texts take a great shrink in size (3 bytes/char → 2 bytes/char). But Korean text in Duncoder keeps the same size (3 bytes/char) as UTF-8.

Abkhazian, Burmese, Central Khmer, Tibetan and Yoruba are rare languages. All of them benefit from Duncoder excludes Yoruba. Because Yoruba are mainly written with ASCII and Latin symbols. Whether a language can benefits depends its character set distribution. Abkhazian are mainly of Cyrillic and Burmese, Central Khmer, Tibetan are of their own character sets. As a result, they achieve huge benefit from Duncoder ([2,3] bytes/char → [1.33,2] bytes/char).

5 Conclusion

In this paper, we introduced Duncode to encode various texts in high space efficiency. It is universal, robust, customizable and keep the unidirectional compatibility with ASCII. With the cost of lookup false positive, Duncode benefits a great space advantage than UTF-8. It is particular suitable for storing diverse texts. In addition, we released a corpus including 179 languages and a tool to collect them. And we built a benchmark for character encoders on all languages.

Acknowledgments

References

[1] G. Graefe and L. D. Shapiro, "Data compression and database performance," *[Proceedings] 1991 Symposium on Applied Computing*, pp. 22–27, 1991.

[2] T. U. Consortium, *The Unicode Standard, Version 13.0.0*. Addison-Wesley Longman Publishing Co., Inc., 2020.

[3] A. F. Ken Whistler, Mark Davis, "Unicode character encoding model," 2008. [Online]. Available: https://www.unicode.org/reports/tr17/#UCS

[4] A. Kumaran and J. R. Haritsa, "- on the costs of multilingualism in database systems," in *Proceedings 2003 VLDB Conference*, J.-C. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, Eds. San Francisco: Morgan Kaufmann, 2003, pp. 105–116. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780127224428500185

[5] M. Mahoney, "Rationale for a large text compression benchmark." [Online]. Available: https://cs.fit.edu/~mmahoney/compression/rationale.html

[6] S. L. Nesbeitt, "Ethnologue: Languages of the world," *Electronic Resources Review*, vol. 3, pp. 129–131, 1999.

[7] V. G. Cerf, "ASCII format for network interchange," *RFC*, vol. 20, pp. 1–9, 1969.

[8] K. Simonsen, "Character mnemonics and character sets," 1992. [Online]. Available: https://www.rfc-editor.org/info/rfc1345

[9] D. Pimienta, "Twelve years of measuring linguistic diversity in the internet: balance and perspectives," 2009.

[10] M. Wolf, K. Whistler, C. Wicksteed, M. Davis, , and A. Freytag, *A Standard Compression Scheme for Unicode*. The Unicode Consortium, Tech. Rep.UTS 6, 2005.

[11] M. Davis and M. Scherer, *BOCU-1: MIME-compatible Unicode compression*. The Unicode Consortium, Tech. Rep. UTN 6, 2016.
[12] D. Salomon, *Data compression - The Complete Reference, 4th Edition.*, 01 2007.
[13] M. Mahoney, "Large text compression benchmark." [Online]. Available: http://mattmahoney.net/dc/text.html
[14] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, pp. 520–540, 1987.
[15] M. Mahoney, "Data compression explained." [Online]. Available: http://mattmahoney.net/dc/dce.html#Section_32
[16] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Comput. Surv.*, vol. 19, pp. 261–296, 1987.
[17] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," *Proceedings DCC '97. Data Compression Conference*, pp. 201–210, 1997.
[18] S. A. of China, *GB 2312-1980: Code of chinese graphic character set for information interchange; Primary set*. China Standard Press, 1981.
[19] M. Dürst, F. Yergeau, R. Ishida, M. Wolf, A. Freytag, and T. Texin, "Character model for the world wide web," 2005. [Online]. Available: https://www.w3.org/TR/charmod/
[20] R. C. Seacord, "Wide-character format string vulnerabilities-robert presents strategies for handling format string vulnerabilities in c." *Dr Dobb's Journal-Software Tools for the Professional Programmer*, pp. 63–65, 2005.
[21] w3techs.com, "Historical yearly trends in the usage statistics of character encodings for websites." [Online]. Available: https://w3techs.com/technologies/history_overview/character_encoding/ms/y
[22] F. Yergeau, "UTF-8, a transformation format of ISO 10646," *RFC*, vol. 2279, pp. 1–10, 1998.
[23] P. Hoffman and F. Yergeau, "UTF-16, an encoding of iso 10646," *RFC*, vol. 2781, pp. 1–14, 2000.
[24] D. A. Huffman, "A method for the construction of minimum-redundancy codes," 1952.
[25] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, pp. 337–343, 1977.
[26] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun.*, vol. 32, pp. 396–402, 1984.
[27] R. Rădescu, "Transform methods used in lossless compression of text files," *ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY Volume*, vol. 12, pp. 101–115, 01 2009.
[28] T. Bell, I. Witten, and J. Cleary, "Modeling for text compression." *ACM Comput. Surv.*, vol. 21, pp. 557–591, 12 1989.
[29] A. Gleave and C. Steinruecken, "Making compression algorithms for unicode text," in *2017 Data Compression Conference (DCC)*, 2017, pp. 441–441.
[30] P. M. Fenwick and S. Brierley, "Compression of unicode files," *Proceedings DCC '98 Data Compression Conference (Cat. No.98TB100225)*, pp. 547–, 1998. [Online]. Available: https://www.cs.auckland.ac.nz/~peter-f/FTPfiles/UnicodeDCC98.pdf
[31] D. Ewell, "A survey of unicode compression," 2004.
[32] S. Atkin and R. Stansifer, "Unicode compression: Does size really matter?" 08 2003.
[33] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," *ACM Trans. Inf. Syst.*, vol. 15, pp. 124–136, 1997.
[34] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates, "Compression: A key for next-generation text retrieval systems," *Computer*, vol. 33, pp. 37–44, 2000.
[35] R. Wan, "Browsing and searching compressed documents," 2003.
[36] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa, "Speeding up string pattern matching by text compression: The dawn of a new era," 2001.