Databases and Information Systems X
A. Lupeikiene et al. (Eds.)

doi:10.3233/978-1-61499-941-6-161

Unrestricted Character Encoding for Japanese

Antoine BOSSARD [a,1] and Keiichi KANEKO [b]
[a] *Graduate School of Science, Kanagawa University*
2946 Tsuchiya, Hiratsuka, Kanagawa, Japan 259-1293
[b] *Graduate School of Engineering, Tokyo University of Agriculture and Technology*
2-24-16 Nakacho, Koganei, Tokyo, Japan 184-8588

Abstract. The glyphs of the Japanese writing system mainly consist of Chinese characters, and there are tens of thousands of such characters. Because of the amount of characters involved, glyph database creation and character representation in general on computer systems has been the focus of numerous researches and various software systems. Character information is usually represented in a computer system by an encoding. Some encodings target specifically Chinese characters: this is the case for instance of Big-5 and Shift-JIS. There are also encodings that aim at covering several, possibly all, writing systems: this the case for instance of Unicode. However, whichever the solution adopted, a significant part of Chinese characters remain uncovered by the current encoding methods. Thanks to the properties and relations featured by Chinese characters, they can be classified into a database with respect to various attributes. First, the formal structure of such a database is described in this paper as a character encoding, thus addressing the character representation issue. Importantly, we show that the proposed logical structure overcome the limitations of existing encodings, most notably the glyph number restriction and the lack of coherency in the code. This theoretical proposal will then be followed by the practical realisation of the proposed database and the visualisation of the corresponding code structure. Finally, an additional experiment is conducted to measure the memory size overhead that is induced by the proposed encoding, comparing with the memory size required by an implementation of Unicode. Once the files are compressed, the memory size overhead is significantly reduced.

Keywords. code, information representation, database, glyph, logogram, symbol, Chinese

1. Introduction

Since the early days of computing, various encoding have been described to address the issue of character representation. In practice, they are databases that include thousands of entries (i.e., characters). When the number of characters involved in a targeted writing system remains low, such as with the Latin alphabet, it is rather easy to describe a corresponding code structure. Obviously, this is completely different in the case of Chi-

An extended abstract of this paper has been published in [1].
[1]Corresponding author; e-mail: abossard@kanagawa-u.ac.jp.

nese characters, and this for two main reasons: first, there is a huge number of glyphs involved, and this number remains unknown, and second, character properties, such as reading, may differ depending on the language or dialect [2].

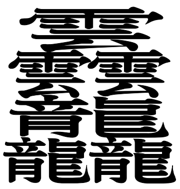
Figure 1. *otodo*.

On current computer systems, numerous Chinese characters are left behind: since not covered by the existing encodings, they are impossible to input, or even simply represent in the system. This is the case for example of the *otodo*, *taito* character given aside in Figure 1 which is notorious for its numerous strokes. This is our aim in this paper to address this essential and critical information representation issue that has plagued Chinese character processing for too long.

Concretely, we shall describe an unrestricted encoding to cover Chinese characters as found in Japanese. The proposed encoding will implement the following key features:

- The collection of an unlimited number of characters.
- High flexibility: glyphs can be added with no code disturbance.
- Ease-of-use: character lookup in the code with minimum effort.
- Character duplicates are not allowed.

While focusing on Japanese, the proposal could be generalised with minor adjustments to all Chinese characters given the essential overlapping of these two character sets.

The rest of this paper is as follows. First, the state of the art is presented in Section 2. Then, we recall in in Section 3 various properties of Chinese characters. Next, we describe in Section 4 the proposed encoding, detailing both the code structure and character lookup methods. Practical work follows with first in Section 5 the realisation of a database that implements the proposed encoding, and the visualisation of the advanced code structure. An empirical evaluation of the memory size overhead induced by the proposed encoding is conducted in Section 6. Finally, this paper is concluded in Section 7.

2. State of the Art

A variety of character encodings are implemented by modern computer systems. These encodings are mainly based on two distinct approaches: the unified approach versus the non-unified approach. In the former method, one encoding is designed to cover all the characters of all writing systems. Unicode [3] is the most well-known encoding that implement this approach. With respect to Chinese characters, this concretely means that the characters used in Chinese, Japanese, Korean, Chu Nom, etc. are merged into the unique encoding, and thus rendered identically whichever the writing system considered. Fonts are sole responsible for the rendering of stylistic differences between the same glyphs of distinct languages.

The unified approach has for obvious advantage that it greatly facilitates the edition of multilingual documents. Nonetheless, the soundness character representation methodology has been much debated over the years; see for example [4]. To begin with, the essential idea of this method, unification, is criticised. How pertinent is it to merge the characters of different cultures, not withstanding the fact that they share some characteristics? In addition, this character merging induces poor code accessibility. It is in fact painfully long to traverse the code in search of a glyph, with a high probability of missing it, since the merged writing systems have their own character ordering and classification methods (for instance based on character reading, number of strokes, etc.).

Even though Unicode's Ideographic Variation Database (IVD, a.k.a. IVS) system [5] enables the declaration of character variants, these additional glyphs remain dependent of font providers and the Unicode consortium's approval (and payment of the registration fee). Moreover, since such characters are defined as variations, they do not appear in the code (i.e., Unicode) and are thus clearly impractical since requiring for example font analysis.

Encodings that target a specific set (or a restricted few) implement the non-unifying approach. Examples include the JIS [6] encoding for Japanese, the Big-5 encoding for Chinese [7] and the EUC-KR encoding for Korean [8]. While the code accessibility in these cases is overwhelmingly higher than that of Unicode – the code being for one writing system can thus be organised homogeneously according to that language's properties – they are severely limited with respect to the number of glyphs represented. Precisely, only a few thousands of Chinese characters are covered by each of these encodings, anyway smaller than or equal to 10,000 glyphs. As a result, the glyphs that are covered are naturally the most frequently used ones, while the others are literally left in the dark. For glyphs to be left aside like this is definitely problematic as computer systems are ubiquitous in the 21st century, and it is well-known that failing to enable the usage of writing systems on them would inevitably result in language extinction.

On the other hand, the IPA *mojikiban* database [9] includes a grand total of about 70,000 characters, which is thus a good indicator of the gap between actual needs and current encoding solutions. Some might argue that the vast majority of the Chinese characters that remain uncovered by current encodings are unused, which is a vicious circle since their absence from computer systems will definitely not help regarding their usage. These characters are perfectly valid, appearing for instance in ancient texts, but not only, and thus deserve to be covered.

For the sake of completeness, it is worth mentioning finally the *Shikaku gōma* (四角號碼) lookup method for Chinese characters [10] which has some relation to the encoding proposed in this paper. This lookup method assigns to each character four or five digits depending on morphological criteria, and thus enables to rather easily locate a character into a dictionary sorted according to these digits. While the *Shikaku gōma* character classification and lookup method has some advantages, it ignores most character properties and therefore is short of structuring. In addition, the character codes induced by the four or five digits are not guaranteed to be unique (i.e., two characters can be assigned the same number), and as a result this method is not suitable as encoding.

3. Definitions and Properties

Terminology is first recalled. The encoding proposed in this paper is focused on the subset of Chinese characters that are found in Japanese. These characters are commonly designated as *kanji*. And to be precise, the Japanese *kokuji* characters are also covered in this work. Character formal identification and unambiguous naming are indeed far from being trivial. Hence, in this paper the code glyphs are simply referred to as (Chinese) characters, and formally gathered in the set $\mathbb{J}$ (additional details on character sets and algebra can be found for example in [2]).

Essential character properties on which the proposed encoding relies are now presented. One should note that only the properties that are required for this work are detailed, and that a more complete ontological discussion is conducted in [2,11].

Radicals Each Chinese character has one unique radical. Relying on radicals is thus a natural way to classify characters, and the conventional method for character dictionaries. In total, there are 214 (modern) radicals.

Strokes A character consists of one or more strokes, which are drawn one by one according to a predefined order.

Variants Some characters have variants, that is, in some cases a character can be drawn in several ways.

4. Information Representation Methodology

In this section, the code structure of the proposed encoding is first described, with then formal tools given for practical code usage.

4.1. Code Structure

Each character is mapped to a unique coordinate in a three-dimensional space. Concretely, the code $\mathcal{C}$ is organised according to a three-dimensional structure with the following three axis.

X axis character radicals;
Y axis number of strokes (not counting the radical ones);
Z axis character variants (if any).

Since the standard 214 character radicals are considered, the X axis spans the integers from 0 to 213, with each radical being assigned a unique index i with $0 \leq i \leq 213$. Radical index assignment is performed conventionally, ordering radicals according to their stroke numbers in ascending order. Formally, let $\hat{\mathbb{J}}$ be the set of the 214 character radicals r_i ($0 \leq i \leq 213$) as defined in Japanese. We define the radical indexing function $r : \hat{\mathbb{J}} \to \mathbb{N}$ that associates to a radical a unique positive integer (index). To this end, we consider the ordered set $\bar{R}$ whose elements are those of $\hat{\mathbb{J}}$, and with the total order relation $r_i < r_j$ for any two distinct radicals $r_i, r_j \in \hat{\mathbb{J}}$ holding if and only if the radical r_i is listed before the radical r_j in the conventional radical ordering of Japanese dictionaries – the sequencing details are abbreviated here, refer for instance to the Kadokawa Shinjigen dictionary [12]. Hence, assuming $\bar{R} = \left\{r_{p(0)}, r_{p(1)}, \ldots, r_{p(213)}\right\}$ in this order for some permutation p of the integers $0, 1, \ldots, 213$, the radical r_i ($0 \leq i \leq 213$) is indexed to $r(r_i) = p(i)$.

For the sake of clarity, the technical details of the Y axis are discussed later in this section. For now, the Y axis simply represents the number of strokes of characters, excluding the radical strokes. For instance, the character 沖 has in total 7 strokes: 3 strokes for the radical (氵), and 4 other strokes (中); the Y coordinate is thus 4. One should note that the radical 氵 is actually a radical variant of the radical 水 (4 strokes), which besides does not impact the way strokes are counted.

The third axis, Z, is used to represent character variants. The character of coordinate 0 on the Z axis is called the "regular" (i.e., standard) variant. Since we are focusing on the Japanese writing system, it is natural to define the regular variant of a character as the one listed in the Ministry of Education's "Table of regular-use Chinese characters" [13], and in general the new form of a character if a new–old distinction is made. In short, we abide by the rules followed by Japanese modern dictionaries. One should note that the radical of a variant is naturally the same as that of the corresponding regular character. Finally, the variants of a regular character are not sorted, that is, for a regular character of coordinate $(x, y, 0)$, the characters of coordinates (x, y, z) with $z > 0$ are arbitrarily sequenced (i.e., unordered).

An example involving the two characters 杉, 枝 both of radical 木 and of stroke numbers (not counting the radical strokes) 3 and 4, respectively, is illustrated in Figure 2.

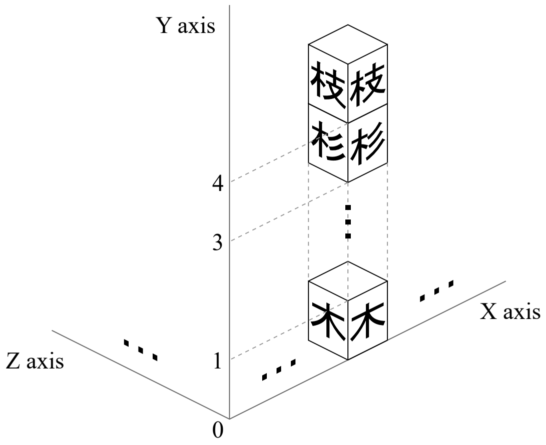

Figure 2. Illustrating the code structure with the characters 杉, 枝 both of radical 木 and of stroke numbers 3 and 4, respectively.

From the above definitions, the following two code properties can be deduced.

Property 1 *Any coordinate* $(i, 0, 0)$ *(*$0 \leq i \leq 213$*) of the code* $\mathcal{C}$ *designates a radical, and conversely.*

Property 2 *A radical of index r may also have variants, thus spanning the* $(r, 0, z)$ *(*$0 \leq z$*) line.*

Obviously, there may exist several characters of a same radical that have the same stroke numbers. So as to address this multiplicity issue, that is with characters possibly colliding on the Y axis, we rely on decimals: the Y axis thus represents the set of the non-negative rational numbers $\mathbb{Q}^{\geq}$. Even though it is possible to further regulate (i.e., order) the affectation of decimals to characters, this would be at the expense of code flexibility: insertion of new characters into the code would be hampered. Hence, decimals are affected to characters of same radicals and same stroke numbers with the only requirement that two characters must not have the same coordinate, which is easily implemented for instance by incrementing the decimal value.

Since there may be more than ten characters of same radical and same stroke number, one single decimal is not enough. Therefore, we fix the number of decimals to 6, which is obviously sufficient given that the overall number of Chinese characters is of the ten thousand order (it might border 100,000, but anyway strictly less than a million). For example, considering the two characters 杉 and 村 which are both of radical 木 and of stroke number 3, their X coordinates are $r(木) = 75$ the index of the radical 木 and their Y coordinates are in the range $[3, 4)$, say for instance 3.000000 and 3.000001, respectively.

Several properties of the proposed code are briefly summarised below. Consider two characters c_1 and c_2 of respective coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2). Assume that they are of same X coordinates, that is $x_1 = x_2$. Therefore,

- they have the same radical;
- if they are both of Z coordinate 0 (i.e., $z_1 = z_2 = 0$), then they are sorted in ascending partial order according to their stroke numbers (i.e., $y_1 \leq y_2$ if and only if c_1 has a stroke number greater than or equal to that of c_2);
- if they both have positive Z coordinates (i.e., $z_1 > 0$, $z_2 > 0$), their respective standard variants (i.e., $(x_1, y_1, 0)$ and $(x_2, y_2, 0)$) are sorted in ascending partial order according to their stroke numbers (i.e., $y_1 \leq y_2$ if and only if the character at $(x_1, y_1, 0)$ has a stroke number greater than or equal to that of the character at $(x_2, y_2, 0)$). In other words, only the characters of Z coordinates 0 are sorted according to their stroke numbers (i.e., according to their Y coordinates).

An illustration in the YZ plane, thus considering characters of one particular radical, is given in Table 1. In this table, "char", "var" and "rad" stand respectively for "character", "variant" and "radical", and the arrow next to the 0 on the Z axis indicates that rows are sorted in ascending order according to the values of this column. Also, by noticing the decimals on the labels of the Y axis as presented previously, one can understand that the characters char2, char3 and char4 have the same stroke number, which is strictly greater than that of char1 and strictly smaller than that of char5.

Table 1. Code structure example in the YZ plane (i.e., the character radical is fixed).

Y axis \ *Z axis*	0 ↑	1	2	3
3	char5			
2.000002	char4			
2.000001	char3	char3 var1		
2.000000	char2			
1	char1	char1 var1	char1 var2	
0	radical	rad var1		

Consequently, a basic lookup function f that partially maps a character to a coordinate in the previously defined three-dimensional space can be defined. This function is used to easily locate one character inside the code (i.e., lookup operation). The function f is said partial as it is surjective. As a prerequisite, we define the stroke number function $s : \mathbb{J} \to \mathbb{N}^*$ which associates to a character its number of strokes (not counting the radical ones). It should be noted that $\forall c \in \mathbb{J} \setminus \hat{\mathbb{J}}, s(c) > 0$, that is, unless a character is a radical, it has at least one stroke in addition to the those of its radical. In addition, let $k : \mathbb{J} \rightarrow \hat{\mathbb{J}}$ be the function that associates a character to its radical.

The function f takes one character or radical as parameter, its domain thus being $\mathbb{J} \cup \hat{\mathbb{J}}$. The codomain of the function f is $\mathbb{N} \times \mathbb{N}$, thus representing two-dimensional coordinates (x, y), with x, y being non-negative integers. Therefore, the function f is used to point at an approximate location in the code, with the interval of rational numbers $[y, y + 1)$ and the Z axis being the approximation range. From this discussion, the basic partial lookup function f is simply defined as $f(c) = (r(k(c)), s(c))$.

Finally, the proposed code features in addition pointers: because two characters may share one common variant, pointers are used to avoid duplicates in the code. For instance, the radicals 邑 and 阜 both have ⻏ as variant. Thus, assuming that the characters 邑 and 阜 have coordinates $(x_1, y_1, 0)$ and $(x_2, y_2, 0)$, respectively, the coordinate $(x_2, y_2, 1)$ designates a pointer which in turn designates the character ⻏ of coordinates $(x_1, y_1, 1)$. Conversely, $(x_1, y_1, 1)$ could designate a pointer which in turn designates the character ⻏ of coordinates $(x_2, y_2, 1)$. This situation is illustrated in Figure 3. A pointer may point at another pointer, in which case the character glyph in the code would be obtained by following the chain of pointers until reaching a non-pointer code element.

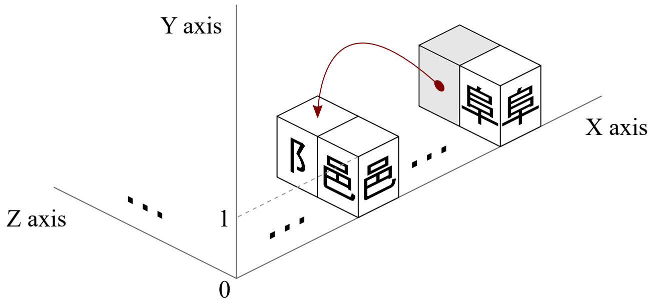

Figure 3. Illustrating pointers in the code: the radicals 邑 and 阜 both have ⻏ as variant; the greyed character block is a pointer.

4.2. Refined Code and Enhanced Lookup Function

Obviously, the more the character properties considered, the more accurate the partial lookup function, which, in other words, is about steering the initial lookup function from surjection towards bijection. In this section, we refine the proposed code – retaining its structure and properties – and its basic lookup function as given previously.

The improvement relates to the Y coordinate of a character. Previously, it was calculated by taking into account the sole number of strokes of the character, thus almost always resulting in the use of decimals in order to distinguish characters of same radicals and same stroke numbers. In addition to the stroke number, we now consider the stroke order, that is the order in which the strokes that make the character are drawn, as well as the types of the used strokes. The former character property (stroke order) is non-ambiguously set by the Japanese government: a character has one unique stroke order. The latter property (stroke types) may be subject to discussion. For the sake of clarity, we restrict the considered stroke types to the basic eight ones as defined for instance by Coulmas [14]: ㇔, ㇐, ㇑, ㇀, ㇏, ㇀, ㇙ (assimilated with ㇚, ㇄, and ㇂) and ㇕ (assimilated with ㇖).

The main idea to implement these two additional character properties into the code without disturbing the code structure is to further rely on decimals as described below.

First, it should be noted that the highest number of strokes for a character in Japanese is 84: this is the character *otodo*, *taito* of Figure 1 mentioned in introduction. Furthermore, only the eight basic character strokes as recalled above shall be considered. Therefore, the stroke order and the stroke types of a character can be represented at the same time by using 84 decimals, say from right to left for the stroke order, with each decimal being in the range 1 to 8 to distinguish between the eight stroke types; the eight strokes are numbered from 1 to 8 in the order they are given above. For instance, the 84 decimals 00…028328 correspond to the character 司. In fact, this character is drawn in the order ㇕, ㇐, ㇑, ㇕, ㇐. It should be noted that the radical strokes are included in the decimals.

To these 84 decimals, it is required to further add say 6 decimals in order to distinguish two characters in the exceptional event that they both have the same radical, stroke number, stroke order and stroke types. The characters 丁 and 丅 are such an exceptional character pair, with thus the 84 decimals not enough to distinguish them: they both induce the 00…032 decimals. Hence, 6 decimals are once again used since this guarantees the possibility of encoding all characters. The 6 decimals are set on the right of the previous 84 decimals so as to retain the code structure previously defined. Therefore, in total, 90 decimals are required in this code enhancement.

As a result, and as stated at the beginning of this section, the Y coordinate of a character has been refined while importantly retaining the previously described code structure: characters are still ordered on the Y axis according to their stroke numbers. In addition, characters are ordered according to their stroke orders and stroke types with the corresponding decimals.

The enhanced lookup function f' can thus be derived as follows. The domain of f' is that of f, that is $\mathbb{J} \cup \hat{\mathbb{J}}$. Unlike f, the codomain of f' is $\mathbb{N} \times \mathbb{Q}^{\geq}$, thus representing two-dimensional coordinates with the Y axis spanning the non-negative rational numbers. Let S be the set of the eight basic character strokes as listed previously. For a character c, assume that $S_c = \{s_0^c, s_1^c, \ldots, s_n^c\} \subseteq S$ is the totally ordered multiset of stroke types such that the character c consists of the $n+1$ strokes s_i^c ($0 \leq i \leq n$), strokes which are drawn in the order $s_0^c, s_1^c, \ldots, s_n^c$. Let $t : S \to \{1,2,3,4,5,6,7,8\}$ be the function that associates a stroke type to its numerical representation (i.e., an integer in the range 1 to 8). Therefore, the enhanced partial lookup function f' is defined as

$$f'(c) = \left(r(k(c)), s(c) + 10^{n-83} \sum_{i=0}^{n} 10^i s_i^c \right)$$

As with the basic lookup function f, the 6 additional decimals for collision handling are not covered by the function.

Because there still exist some extremely rare cases where two characters have the same radical, the same stroke number, the same stroke order and the same stroke types – it was indeed hard for the authors to exhibit one such character pair example – the refined lookup function remains surjective. Therefore, even though in most cases the function will directly point at the character being looked up, there may be some possibility that it does not, in which case the function is pointing at a restricted area in the code – actually a very restricted area now that the lookup function has been refined.

In other terms, we have defined a hash function for any Chinese character as used in Japanese. This hash function is a non-perfect one as there remains a few (extremely) rare cases where two distinct characters are hashed to the same value. But because of the extreme rarity of such colliding characters, and obviously of the extreme morphological (rendering) similarity of the two characters, this hash function can be safely used for the conventional hashing purposes: database, cryptography, etc.

5. Database Realisation and Code Visualisation

An important objective of the proposed encoding is an improved accessibility. In order to demonstrate this property, a large character database is implemented and the proposed code is illustrated by means of visualisation. As the introduced code is based on a three-dimensional structure, we naturally rely on 3D graphics to illustrate this spatial characteristic of the database.

The data used for the implementation of the proposed encoding and the corresponding visualisation system has been obtained from the Japanese governmental Information-technology Promotion Agency (IPA): this is the *mojikiban* (文字情報基盤) character database [9], including nearly 70,000 entries. Precisely, our implementation includes 56,875 characters of the IPA database. As the character stroke information included in the IPA database is limited to the stroke number, thus including neither the stroke order nor the stroke types, the described code enhancements (see Section 4.2) are absent from this implementation. Expansion of the IPA database to include such missing information, and subsequent updates to the code implementation are part of future work. As an implementation note, it is finally mentioned that in the case the database mentions several radicals for one character, the first radical was retained; refer for instance to [2] for additional details on this issue. Besides, this radical plurality explains the low number of characters in the database for a few radicals such as 匸).

This database implementation based on the proposed code structure as well as the corresponding visualisation system were realised on an Intel i7-6700 CPU, 16 GB RAM machine running a 64-bit Windows 10 operating system. The database was built in two steps as follows. First, selected character information such as radical and stroke number was retrieved from the IPA database and compiled into a flat text file for fast database loading. This process takes about 2 hours 30 minutes on this experiment machine since requiring multiple traversals of the IPA database. Importantly, this first step is executed only once: the IPA database is not used afterwards.

Second, the resulting file is loaded into memory and the proposed database structure realised in two passes (so as to address the memory allocation issues). This process is completed in the order of a minute, which is very fast considering the number of characters involved – this loading time is to be compared with, for example, the time required for the first step. The program takes about 260 MB of memory at run-time, mostly due to the textures generated for every glyph. Thanks to this optimised approach, the visualisation (rendering) of the assembled database remains smooth at all time, enabling seamless navigation (e.g., zoom, camera movement) on portions of the rendered code, thus providing a high accessibility for the code as glyphs can be easily located as previously explained.

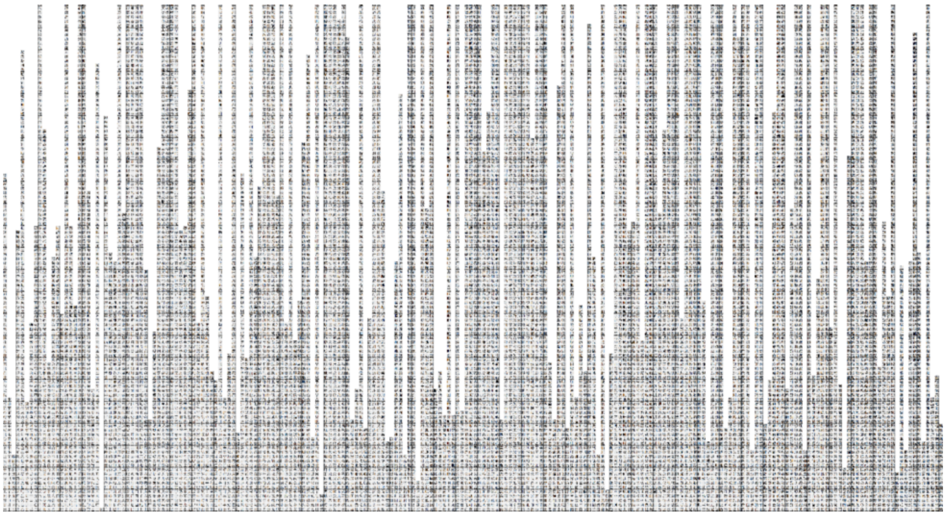

Figure 4. Overview in the XY-plane of the assembled database (only cut at the top), from the first radical (left) to the last one (right).

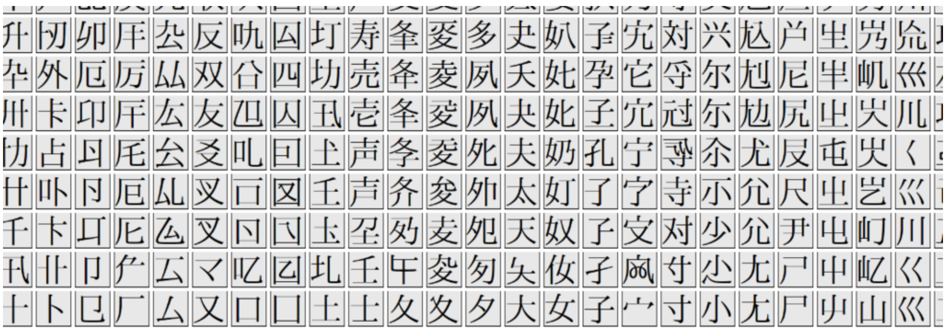

Figure 5. Zoom-in on an excerpt of the assembled database.

Several sample illustrations are given below. First, an overview in the XY-plane of the assembled database is shown in Figure 4. Then, a more detailed view that shows characters in a readable form is given in Figure 5. Next, sample characters on the Z axis, that is non-regular character variants, are shown in the “rear” view given in Figure 6.

Finally, the repartition of the regular characters (i.e., the characters of coordinate $z = 0$) considering radicals, and that of the non-regular characters (i.e., the characters of coordinate $z > 0$) are given for reference in Figures 7 and 8, respectively. These repartitions show character counts for each of the 214 radicals. It is especially interesting to see with Figure 8 that the repartition of non-regular characters is not uniform, most notably with the radical 艸 including in total 894 non-regular characters.

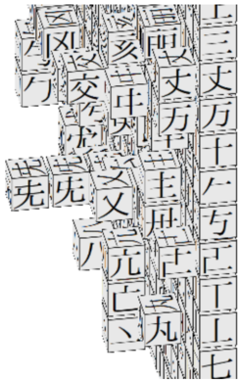

Figure 6. A detailed view on characters on the Z axis (i.e., non-regular character variants).

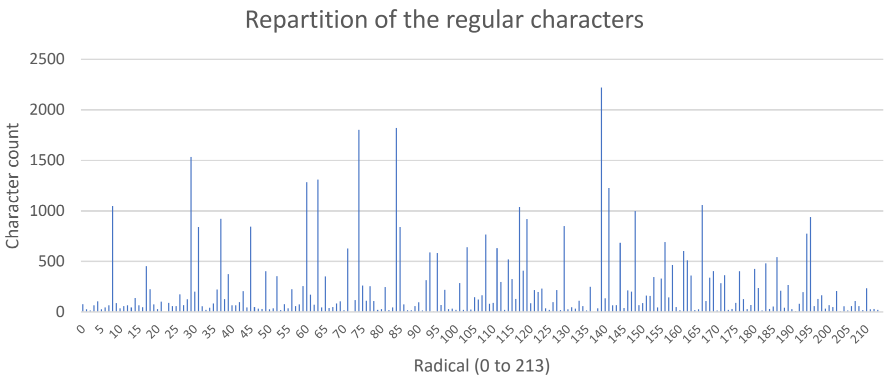

Figure 7. Repartition of the regular characters (i.e., of coordinate $z = 0$) considering radicals.

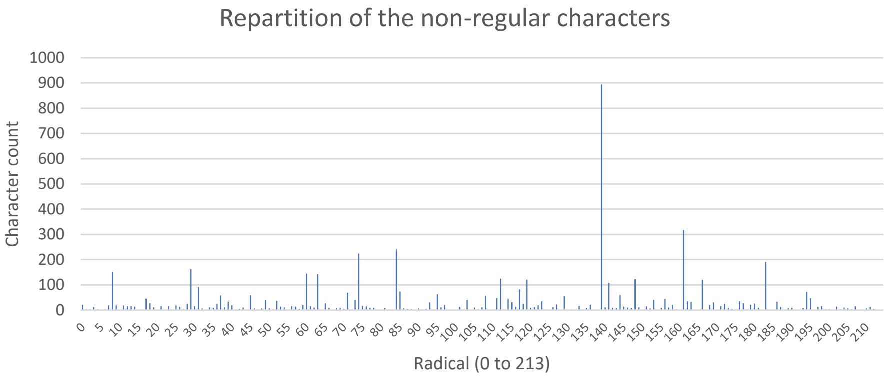

Figure 8. Repartition of the non-regular characters (i.e., of coordinate $z > 0$) considering radicals.

6. Empirical Evaluation: Memory Size Overhead

In this section, we empirically evaluate the overhead induced by the proposed encoding when representing Chinese character strings in memory. This experiment relies on the encoding implementation as detailed in Section 5 and has been similarly conducted on the same environment (Windows 10, 64-bit operating system).

More precisely, we shall compare in this experiment the occupied memory size of two identical strings when stored with the proposed encoding and with the UTF-16 implementation of Unicode, which is standard for the Windows API. Once again and for the same reason, this experiment does not include the code enhancements as detailed in Section 4.2.

First and foremost, it is required to define a binary convention for character storage in memory. This can be directly deduced from Section 4: the X coordinate of a character is included in the integer range 0 to 213 and thus stored with one single byte. The Y coordinate is a rational number of at most 6 decimals and whose integral part is at most 84 (i.e., the highest number of strokes for a Japanese character as explained). Hence, according to the IEEE 754 floating-point standard, 27 bits are required in the mantissa (i.e., bit exponents ranging from +6 to -20). Since a single precision float (32-bit) has only 23 bits in the mantissa, a double precision float (64-bit) is required; it has a 52-bit mantissa, which suffices in our case. Finally, regarding the Z coordinate, which is an integer, the maximum number of variants for one representative character in our database is 15 (i.e., at most 16 characters on the Z axis for any X, Y coordinates). Hence, the Z coordinate is just the X coordinate stored as one single byte. Obviously, this convention can be adjusted upon needs. This binary convention is summarised in Table 2 and examples, including variants, are given for reference.

Table 2. Binary convention for character storage in memory. The representation of several characters is given as example (in hexadecimal notation).

Size	1 byte	8 bytes	1 byte
Description	X (radical)	Y (representative character)	Z (variant)

Examples:

Character 亀	`0x04`	`0x4024000064A9CDC4`	`0x00`
Character 乳	`0x04`	`0x401C000000000000`	`0x00`
Character 乳	`0x04`	`0x401C000000000000`	`0x01`

So as to empirically evaluate the memory size overhead induced by the proposed encoding, each of the first 1024 characters of the database has been processed as follows. First, the character's X, Y and Z coordinates are calculated according the proposed encoding. Then, these coordinates are converted in binary according to the previously described binary convention (see Table 2). Next, the obtained binary sequence is written on disk, concretely appended at the end of the file `fileE`. Hence, `fileE` consists in a sequence of 10-byte records, one per character. For comparison, the Unicode (UTF-16) sequence corresponding to the character is then generated, and the obtained binary sequence is written on disk, concretely appended at the end of the file `fileU`. As an illustration, the first 32 bytes of the two files are given in Table 3. Colours are used to match and separate character bytes.

Table 3. The first 32 bytes of the generated files corresponding to the proposed encoding (i.e., `fileE`) and the UTF-16 Unicode implementation (i.e., `fileU`). Character bytes are matched and separated with colours.

	Proposal (i.e., `fileE`)	**Unicode (UTF-16)** (i.e., `fileU`)
`0x000`	`02 00 00 00 00 00 00 00`	`05 30 06 30 3B 30 00 34`
`0x008`	`40 00 03 00 00 00 00 00`	`01 34 02 34 04 34 40 DB`
`0x010`	`00 F0 3F 00 02 00 00 00`	`01 DD 04 34 40 DB 00 DD`
`0x018`	`00 00 00 F0 3F 00 00 00`	`05 34 06 34 0C 34 16 34`
`0x020`	`...`	`...`

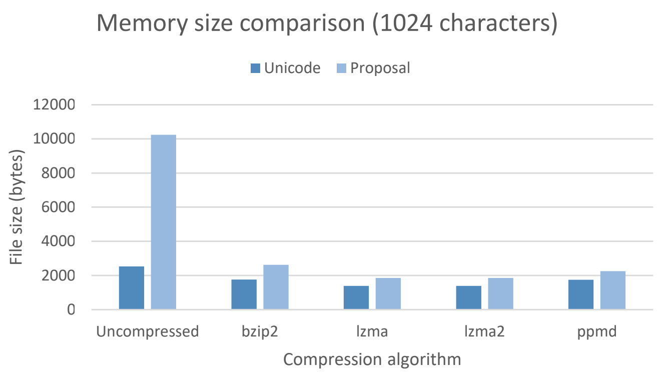

Figure 9. Empirical evaluation of the memory size overhead induced by the proposed encoding.

The memory size overhead has been subsequently measured as follows: file size measurement for the uncompressed files (`fileE` and `fileU`), and file size measurements for the files when compressed with various algorithms (namely, the bzip2, lzma, lzma2 and ppmd compression algorithms). The results are given in Figure 9. From these results, it can be noticed that while the uncompressed file size of `fileE` is obviously – since our proposal is a higher-expectation encoding – larger than that of `fileU`, the file size gap is significantly shrunk when applying whichever of the tested compression algorithms. Nevertheless, one can note that the lzma compression algorithm produces the best results: the file size is lowest, the compression ratio is highest (82%), and while not the smallest, the overhead compared with Unicode is at 34%. Additional details are given in Table 4. This very good compression ratio is mostly due to the occurrence of sequences of 0 bytes as illustrated in Table 3 (this is to be compared with the Unicode file). These 0 sequences are induced by the decimals of the Y coordinate.

Table 4. Details of the memory size measurements. File sizes are given in bytes. The overhead percentage is calculated with the Unicode file size as reference.

		Compression algorithm			
Encoding	**Uncompressed**	bzip2	lzma	lzma2	ppmd
Unicode (UTF-16)	2,530	1,760	1,386	1,393	1,748
Proposal	10,240	2,620	1,851	1,858	2,250
overhead	(+305%)	(+49%)	(+34%)	(+33%)	(+29%)
compression ratio		(74%)	(82%)	(82%)	(78%)

7. Conclusions

In this paper, a formal database structure for Chinese characters in the form of a character encoding has been proposed. To the difference of previous works such as Unicode and the IPA *mojikiban* database, the described model has been designed to rely on, and retain as much relationship information between entries (i.e., characters) as possible. As a result, the proposed code is significantly easier to use, for example for searching operations. In addition, the proposed encoding remains flexible by allowing the addition of new glyphs when necessary, and this without any code disturbance (i.e., modifying the code mapping). And, the number of characters covered is not limited as in other encodings. Besides the formal definition of the proposed encoding, we have shown as a proof of concept how to concretely build such a database, providing a three-dimensional visualisation of the code structure to illustrate the spatial characteristic of the realised database and the induced high accessibility. Finally, we have conducted another experiment to empirically evaluate the memory size overhead that is induced by the proposed encoding, comparing with an implementation of Unicode. The obtained results showed that once compressed, the memory size overhead was significantly reduced to (e.g., less than 29% overhead with the ppmd compression algorithm).

Regarding future works, the inclusion of stroke order and stroke type information in the assembled database is meaningful so as to implement the enhanced lookup function. In addition, further refining the initial binary representation that was proposed in this paper is an important future work with respect to the practicability of the presented encoding for current computer systems. As a refining strategy, the stroke type and order information would first need to be included into the binary representation, before considering the optimisation of this representation.

Acknowledgements

The authors sincerely thank the reviewers for their insightful comments and suggestions. This research project is partly supported by The Telecommunications Advancement Foundation (Tokyo, Japan).

References

[1] Bossard, A., Kaneko, K.: Proposal of an Unrestricted Character Encoding for Japanese, Proceedings of the 13th International Baltic Conference on Databases and Information Systems, Communications in Computer and Information Science 838, 189–201, Trakai, Lithuania, July 2018.

[2] Bossard, A.: Chinese Characters, Deciphered. Kanagawa University Press, Yokohama, Kanagawa, Japan (2018).

[3] The Unicode Consortium: The Unicode Standard 5.0. Addison-Wesley, Boston, MA, USA (2007). More recent versions accessible online at http://www.unicode.org/versions/latest/.

[4] Sekiguchi, M. (Fujitsu): 標準化教育プログラム - 第12章 文字コード標準 (in Japanese). Japanese Standards Association (JSA) (2006).

[5] Lunde, K., Cook, R., Jenkins, J. H.: Unicode Ideographic Variation Database, Unicode Technical Standard no. 37, version 5.0 (2018). http://www.unicode.org/reports/tr37/. Last accessed September 2018.

[6] Japanese Industrial Standards Committee (JISC): 7-bit and 8-bit Coded Character Sets for Information Interchange (7 ビット及び 8 ビットの情報交換用符号化文字集合, in Japanese). (1969).

[7] Lunde, K.: CJKV Information Processing. O'Reilly Media, Sebastopol, CA, USA (2009).

[8] Choi, U., Chon, K., Park, H.: Korean Character Encoding for Internet Messages (Request for Comments #1557, Network Working Group). Internet Engineering Task Force, Fremont, CA, USA (1993). https://tools.ietf.org/html/rfc1557. Last accessed March 2018.

[9] Information-technology Promotion Agency (Japan): *Mojikiban* Database (文字情報基盤文字情報一覧表, in Japanese). http://mojikiban.ipa.go.jp/. (2016). Last accessed February 2018.

[10] Morohashi, T.: Daikanwa Jiten (大漢和辞典, in index). Taishukan Publishing, Tokyo, Japan (2007).

[11] Bossard, A., Kaneko, K.: Chinese Characters Ontology and Induced Distance Metrics, International Journal of Computers and Their Applications 23(4), 223–231 (2016).

[12] Ogawa, T., Nishida, T., Akatsuka, K. (editors): Kadokawa Shinjigen (角川 新字源, in Japanese), revised version. Kadokawa, Tokyo, Japan (1994).

[13] The Agency for Cultural Affairs, Japanese Ministry of Education, Culture, Sports, Science and Technology (MEXT): Table of the Regular-use Kanji Characters (常用漢字表, in Japanese). (2010).

[14] Coulmas, F.: The Writing Systems of the World. Basil Blackwell, Oxford, England (1989).