Optimizing Heisig's method for learning Japanese and Chinese characters

Oscar Cabrera

ocg.steppenwolf@gmail.com

Independent Researcher

November 2025

Abstract

We will analyze Heisig's method, which he created to build by hand sorted lists of Japanese and Chinese characters, in order to apply it in a pure and systematic way to obtain better and easily adaptable sorted lists in response to changes in the character lists, making learning easier for the student and avoiding degradation in the sorting.

1 Introduction

James W. Heisig is renowned among students of the Japanese and Chinese languages for his book series *Remembering the Kana* [1], *Remembering the Kanji* [2][3][4] (also known as RTK1, RTK2 and RTK3), and *Remembering the Hanzi* [5][6][7][8].

All the characters are analyzed by components, called *primitives* by Heisig, which may be traditional radicals, a collection of strokes or other characters. These primitives are introduced as needed. Each character (and each primitive) is assigned a unique keyword. A character's written form and its keyword are associated by imagining a story connecting the meaning of the character with the meanings of all its used primitives. For instance, the character *four* 四 is constructed from the primitives *enclosure* 冂 and *human legs* 儿.

In the sorted lists developed by Heisig in his books, there are times when incoherencies occur: a character is in a misplaced location because it is assigned an inappropriate primitive or a primitive that has not yet been shown; a character should have been shown earlier; *ad hoc* decisions grouping characters; etc.
Although these problems can be fixed by hand, it also makes you wonder if the final sorted list of primitives and characters is the most appropriate for the student, because what we want when introducing a new primitive or character is to show below all the primitives or characters that are displayable.

The objective of this text is to use Heisig's method in a pure, unsupervised and systematic way, without making exceptions, *ad hoc* decisions or incoherencies. In addition, this way we will avoid the degradation in the quality of an existing sorting when it is necessary to adapt it to new changes that occur in the character lists. Since Heisig did not explicitly formalize his method, we will attempt to analyze it from the sorted lists he made in his series of books.

2 Analysis

Heisig's method is applied to the following problem, which we refer here as *Gaikokujin no chōsen* (外国人の挑戦, or *The foreigner's challenge*): a non-native student wants to learn how to correctly write a given list of Japanese or Chinese characters, as well as a unique keyword for each character. How can we sort the list to make it easier for students to learn? We assume here that the first time a character is introduced is crucial for the learning, so it doesn't matter if the order of the list is not followed in subsequent reviews by using flashcards in random order.

Demonstrating whether Heisig's method is the best way to approach this problem is beyond the purpose of this text, since the problem has a strong subjective component: what do we mean by better or easier learning? It is rather a question for neuroscience. Then, we will assume that a constructive method like Heisig's is suitable for making things easier for the human brain during learning.

Let's start with the block diagram for Heisig's method (Figure 1). Before explaining each module, it is understood that the information for each primitive and character must be stored in some kind of database that we carefully created beforehand; that hidden stage will be responsible for most of the subjectivity in the application of the method.

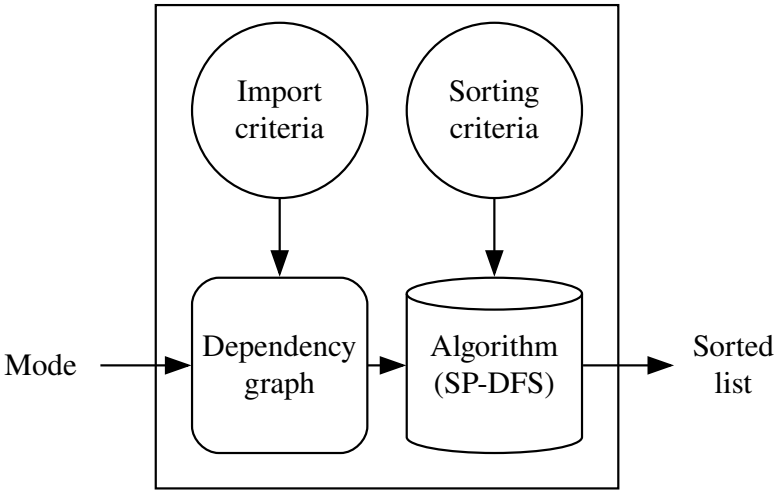


Figure 1: Block diagram for Heisig's method

2.1 Input and output

In the *Mode* we indicate the sets of nodes to be loaded, specifying the load type for each one (see section 2.2) in order to generate the dependency graph. Keep in mind that we must include every set whose nodes are directly or indirectly used.
A node (character, primitive, etc.) may belong to several sets at once, what makes it interesting to generate sorted lists for different modes. For instance, character *four* 四 may belong to sets Jōyō kanji, RTK1, and others.

The *Sorted list* is simply the output of the algorithm: a sorted list of characters and primitives, according to the chosen mode.

2.2 Import criteria

In the *Mode* we have specified the load type for each set we want to load. Heisig handles three kind of loading for the sets:

- Normal
  The characters and primitives of these sets are those we want to sort.
- Learned
  The nodes of these sets will be considered learned or seen, and therefore, won't be shown.
  For example: when a character of RTK3 has as primitive another character of RTK1, that primitive is considered learned and is not displayed.
- Imported
  The nodes of these sets won't be considered as learned. When a character or primitive belongs to an imported set, the node may be either shown as primitive or not shown at all, according to certain criteria.
  For example: when a character of RTK1 has as primitive another character of RTK3, that primitive may be displayed or not, depending on the case.

The only subjective parameter in these criteria is the minimum number of child nodes that an imported character must have to be shown as a primitive. Appropriate values would be 1 to 3, depending on how strict we want to be (Heisig usually uses a value of 2 or 3).

2.3 Dependency graph

Nodes in loaded sets may have dependencies on other nodes. For example, in the case already mentioned, we would add a character node to the graph for *four* 四 and two primitive nodes for its dependencies *enclosure* 囗 and *human legs* 儿 (Figure 2). It means that node 四 can't be shown before showing nodes 囗 and 儿. Besides, 四 can act as a primitive for other nodes.

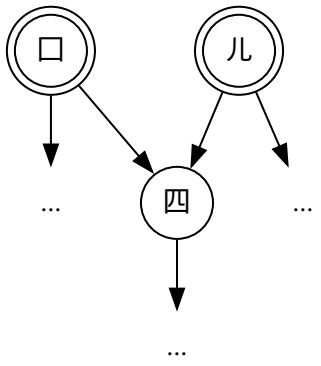


Figure 2: *Four* 四 (single circles for characters) has two dependencies (double circles for primitives): *enclosure* 囗 and *human legs* 儿

For each node, it is enough (and recommended) to indicate the major dependencies to build that node, without indicating all the possible dependencies which are already implicit in the others.

Then, from the loaded sets and an internal node called *root*, we will obtain our dependency graph, which deserves a formal definition:

Definition 2.1. *The dependency graph used in Heisig's method is a directed graph or digraph* $G_t = (V, E_t)$ *at instant* t*, where:*

- V *is a set of vertices (or nodes), where* $root \in V$*.*
- $E_t \subseteq \{\{a, b\} \mid a, b \in V \wedge a \neq b\}$*, a set of edges (or links) connecting two distinct vertices at instant* t*.*
- G_t *must be a simple graph: a graph without loops or multiple edges between two nodes.*
- G_t *may handle edge labeling, but its utility depends on the implementation.*
- G_t *and* E_t *may change over time, due to the action of the sorting algorithm.*
- G_0*, the initial graph, must be weakly connected: replacing all of its directed edges with undirected edges produces a connected (undirected) graph. It will always be true because of Proposition 2.3.*

More definitions are required:

Definition 2.2. *Given a node* $n \in V$*:*

- $P_t(n)$ *is the set of parent nodes for* n *at instant* t*. In other words, those nodes are the dependencies of* n*.*
- $C_t(n)$ *is the set of child nodes for* n *at instant* t*. That is to say,* n *is a dependency for those nodes.*
- $A_t(n)$ *is the set of ancestor nodes for* n *at instant* t*.*
- $D_t(n)$ *is the set of descendant nodes for* n *at instant* t*.*

Definition 2.3. X_t *is the set of explorable nodes at instant* t*.*

To ensure that, when loading the sets, there are no floating nodes disconnected from the graph, we will apply the following proposition:

Proposition 2.1. $G_0 = G_0 \cup \{\{root, n\}, \forall n \in V : |P_0(n)| = 0\}$*. It means that every node without dependencies will be assigned a dependency to the root node.*

Therefore, we can state the following propositions:

Proposition 2.2. $root \in A_0(n), \forall n \in V$*. This is a consequence of Proposition 2.1.*

Proposition 2.3. G_0 *is a directed rooted tree. This is a consequence of Proposition 2.2.*

Finally, as a big simplification we can state that Heisig's method is based in the following axiom:

Axiom 2.1. *Given a node* $n \in V$*:* $|P_t(n)| = 0 \iff n \in X_t$

This axiom is a matter of common sense: it is necessary to explore all parent nodes of n before exploring n itself, assuming that this constructive point of view is the most pedagogical for the student, as we already discussed.

Take into account an important subjective factor: two persons may assign different dependencies for certain nodes and even use a different number of node types, so at the end they will have dependency graphs with a different structure. A well-structured dependency graph is crucial to achieving a good sorting.

2.4 Algorithm (SP-DFS)

Once the dependency graph has been built with all the nodes for the loaded sets, the exploration of this graph will give us a list with the visited nodes, and therefore, the sorted list of the primitives and characters to be displayed.

Initially, since the root node is the only explorable node, according to Axiom 2.1, we will start the exploration from that node. So we can now create the basic implementation for the proposed block diagram (Algorithm 1).

Algorithm 1 Basic algorithm to sort the list

Input: $Mode$
Output: $SList$
1: $SList \leftarrow [\,]$ ▷ Initialize sorted list
2: $G \leftarrow$ `CreateDepGraph`$(Mode)$ ▷ Import criteria are applied
3: $SList \leftarrow$ `SP_DFS`$(G, \texttt{root}, SList)$ ▷ Sort nodes starting from root, using sorting criteria
4: **return** $SList$

We will call the exploration algorithm *Sort&Prune-DFS* (SP-DFS), because it is based in the well-known DFS algorithm (Depth-first search) [9]. Remember that DFS is an algorithm for exploring trees and graphs: it starts at the root node and explores as far as possible along each branch before backtracking; it uses a stack (or recursion) to keep track of the nodes to visit. Our SP-DFS algorithm also performs a depth-first search, but has some differences compared to the classic DFS:

- A node is only visited when it is explorable (Axiom 2.1), that is to say, in the last visit.
- When a node is explored:
    1. The edges to all its child nodes are removed. This means that, since the node has already been explored, the dependencies of its children on that parent node disappear.
    2. Its child nodes are sorted to select the *best* node to continue the exploration. The sorting must be recalculated each time the algorithm comes back to the explored node (backtracking) to continue exploring its children, because the exploration of a child may potentially affect the ranking of other children in the sorting.

Theorem 2.1. *The SP-DFS algorithm is always able to explore the entire dependency graph G_0 from the root node.*

Proof. Since each node in our graph G_0 has the root node as a common ancestor (Proposition 2.2), the SP-DFS algorithm will visit all the nodes because:

1. It is a known fact that, by the principle of mathematical induction, DFS correctly visits every node in a tree. The recursive nature of the algorithm combined with backtracking ensures that each node is explored and its entire subtree is visited before moving to the next sibling or returning to the parent.
2. The basic exploration mechanism of SP-DFS is identical to that of DFS; neither the different order in node exploration nor the edge removal don't affect the possibility of a node to be explored, only the moment it is explored.
3. According to Proposition 2.3, G_0 is a tree.
4. Then, the proof for DFS is also applicable to G_0.

□

The previous theorem tells us that all the nodes in the graph are potentially explorable by the SP-DFS algorithm, but keep in mind that, in practice, the algorithm may decide not to explore all of them to avoid incoherencies; this will depend on the specific implementation. Take into account that we're not going to prove that Heisig uses this algorithm in his method; as we said, it's what emerges from reading his books.

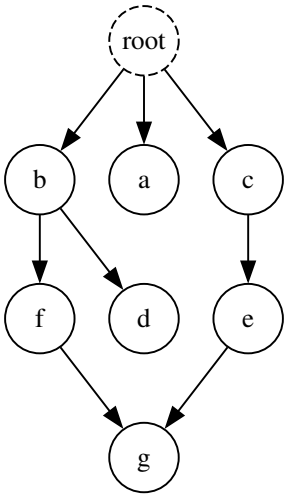
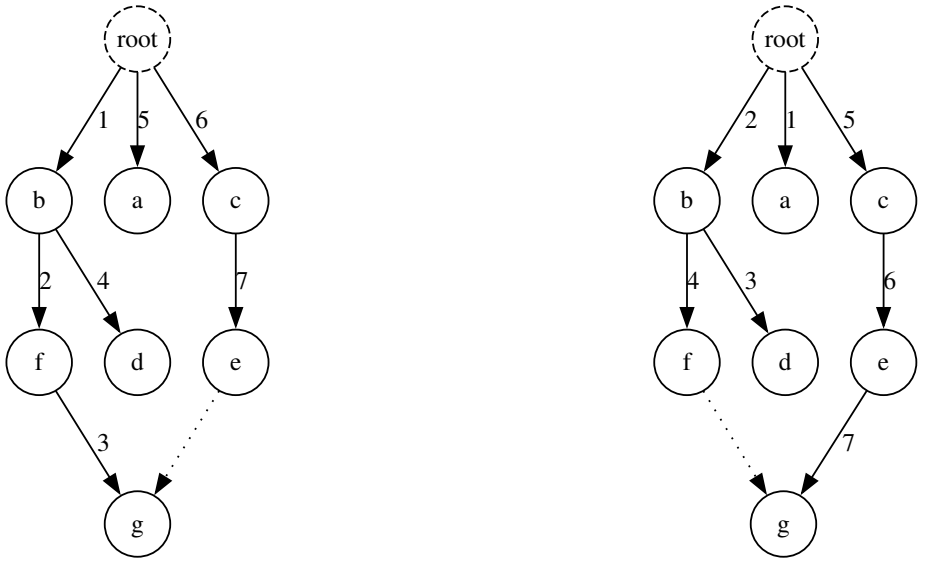


Figure 3: Example of dependency graph to explore from the root node

Well, let's look at a simple example of exploring a graph starting from the root node (Figure 3). Let's compare the generated sorted list depending on the algorithm we apply to explore the graph:

- DFS (Figure 4a): root, b, f, g, d, a, c, e. We have assumed that left edges are chosen before right edges, and the search will not repeat previously visited nodes.
- SP-DFS (Figure 4b): root, a, b, d, f, c, e, g. We have used lexicographic order as the only sorting criterion. Pay special attention to node g:
  1. When exploring its parent node f, all edges to its child nodes are removed (in this case, its edge $\{f, g\}$), but we cannot continue exploring node g because it does not satisfy Axiom 2.1 (i.e. $g \notin X_t$).
  2. Once we visit node e, we can now explore node g because it has no more parents after removing edge $\{e, g\}$ (i.e. $g \in X_t$).



(a) DFS **(b)** SP-DFS

Figure 4: Comparing two types of graph exploration (dotted edge is not followed)

We present a recursive version of SP-DFS (Algorithm 2); apart from the differences already mentioned with respect to DFS, some considerations need to be made:

1. The recursive function SP-DFS appends nodes to $SList$, so at the end of the graph exploration we will have there the full sorted list of nodes.
   The value returned by the function is indicated for clarity only; since $SList$ is an input and output value, in practice it should be implemented by passing this value by reference, so the function should not actually return anything explicitly.
2. The edge removal (the first if clause) is done inside the while loop, not outside; this is because, although it will usually run once, when $Node$ is the root node it may need to run more than once due to the extra child nodes added dynamically to the root during the recursive process.
3. If a child node is explorable, the algorithm calculates a weight structure for this child; these weights will be taken into account by the insertion of the child node in *snodes*.

4. The *snodes* variable must be a data structure that uses its own comparison function in order to evaluate the sorting criteria each time a new element is inserted.

5. As a preventive measure, we validate the child node to be inserted in order to obtain a good sorting. Normally, the validation will be successful, but there are some cases where it may not be. For instance, in the following trivial case: suppose we are exploring a node a, which has two child nodes: a character b and a primitive c (Figure 5).

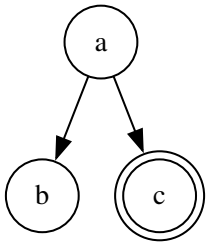


Figure 5: Example of unsuccessful validation exploring from node a

   In this case, although both child nodes are perfectly explorable (after a is explored), node c should not pass the validation as it would be odd to display a primitive without displaying some characters that use it.

6. If a child node cannot be explored at a given time, it is possible that we may be able to explore it later.

7. After sorting the child nodes, we see if there are any nodes in the sorted list *snodes*.

   (a) If there are none, for each child we will create a dependency with the root, so that they do not disconnect from the graph (if our implementation uses edge labeling, we will assign a default label to the new edge).
   Finally, we exit the loop of sorting.

   (b) Otherwise, we remove the first child node from the list of explorable child nodes and continue recursively exploring that node (i.e. the *best* one, in terms of sorting).

8. Note that, at the end of the global exploration, the remaining child nodes of the root node are the problematic nodes in the graph (neither explored nor validated), possibly due to a bad structure of nodes in the data sets, and they should be manually reviewed and fixed.

This module is almost entirely objective; the subjective perception of how good or bad the final sorted list may be depends on how well we have implemented the sorting criteria, the dependency graph, and the validation function.

Algorithm 2 Recursive function `SP_DFS`

Input: G, $Node$, $SList$
Output: $SList$

```
1:  SList.Append(Node)                                   ▷ Append Node to sorted list
2:  children ← [ ]                                       ▷ Initialize explorable children of Node
3:  while children.size() > 0 ∨ |C(Node)| > 0 do         ▷ Sort explorable children of Node?
4:      if |C(Node)| > 0 then                            ▷ Is edge removal necessary?
5:          for each c ∈ C(Node) do                      ▷ Remove dependencies of Node for every child c
6:              G.RemoveEdge(Node, c)                    ▷ Remove edge {Node, c}
7:              if |P(c)| = 0 then                       ▷ Is c explorable? (see Axiom 2.1)
8:                  children.Insert(c)                   ▷ Store explorable child node
9:          if children.size() = 0 then
10:             break                                    ▷ No more explorable child nodes, exit the while loop
11:     snodes ← [ ]                                     ▷ Initialize sorted validated children
12:     for each c ∈ children do                         ▷ Sort the children of Node, in snodes
13:         c.weights ← G.CalcWeights(c)                 ▷ Compute weights for child c
14:         if G.Validate(c) = True then                 ▷ Is child c really explorable?
15:             snodes.Insert(c);                        ▷ Insert validated child node into the list
16:     if snodes.size() = 0 then                        ▷ No child nodes in sorted and validated list?
17:         for each c ∈ children do                     ▷ Reconnect child nodes to the graph via root node
18:             G.AddEdge(root, c)                       ▷ Add edge {root, c}
19:         break ▷ No more explorable child nodes able to be validated, exit the while loop
20:     children.Erase(snodes[0]) ▷ Delete the selected child node from explorable children
21:     SList ← SP_DFS(G, snodes[0], SList)              ▷ Continue exploring the selected child node
22: return SList
```

2.5 Sorting criteria

These criteria are used by the SP-DFS algorithm to sort the list of explorable child nodes of a given node. These criteria have a strong subjective factor, because they depend on the characteristics of the nodes: types of nodes, edge labels, number of strokes, number of dependencies, weight of the node, etc. To achieve a good sorting, a careful selection of these criteria is necessary. They can be easily deduced from Heisig's books, and even improved.
As an example of criterion: Heisig generally prioritizes the display of characters rather than the introduction of new primitives (when possible).

2.6 Complexity

In order to calculate the global complexity of the method, take into account that the input of the algorithm is not the *Mode*, but the dependency graph (Figure 1).
Therefore, the space complexity is $O(|V| + |E|)$ due to storing the graph using an *adjacency list*. The average time complexity depends on the graph structure and the average cost to calculate the weights of an explorable node (function $w(V, E)$), but an estimation would be

$O(|V| + \frac{|E|^2}{|V|}w(V,E))$, where $w(V,E)$ strongly depends on the implementation. In general, it's reasonable to say that $O(1) \leq O(w) \leq O(|V| + |E|)$, but a good value in practice should be logarithmic.
Anyway, the time complexity doesn't matter too much because it's not an algorithm that needs to be executed intensively, but rather sporadically to create the sorted lists.

2.7 Optimization

Note that, by applying Algorithm 2, we obtain the *first* valid sorting that satisfies Heisig's method: import criteria, dependency graph, and sorting criteria. But, in fact, there may be more than one valid sorting... are they all equally good? One way to evaluate this is to calculate the performance of each valid sorting, and keep the *best* one (there may also be more than one). It may seem like an intractable problem, but it probably won't be, because we don't have to generate all possible sortings (valid or not), but only the valid sortings directly.

For this purpose, it is easy to make a small change to Algorithm 2 so that it finds the first among all the best valid sorted lists. In principle, there won't be many of them, so the impact on time complexity will not be significant... but it depends on the structure of the graph, how restrictive the sorting criteria are, and how we calculate the performance.

The way we develop the performance calculation is somewhat subjective and has a great influence on what we understand as a *valid sorting*, but here we will propose a fairly logical and intuitive one based in the size of the character intervals between two primitives. Let's think about the two extreme cases (not realistic, since they would not be valid sortings):

- The worst theoretical case is that we put all the primitives at the beginning and then all the characters.
- The best theoretical case is that we distribute the characters equally in the intervals (since we are evaluating the best case, we will assume an additional interval without an initial primitive).

To facilitate student learning, it seems desirable to have intervals of similar size, whenever possible, always complying with Heisig's method. Then we will understand that, to maximize performance, we must minimize the RMS value of the interval size:

$$x_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{1}$$

Here, N is the number of character intervals, and x_i is the number of characters in the interval i. Intervals are always bounded by primitives, except for the first interval, which may or may not have a primitive before it. Then, if the algorithm generates all valid sorted lists (let's say M), we can compute the real extreme values for x_{RMS}:

$$x_{RMS_{best}} = \min_{j \in [0..M-1]} \{x_{RMS_j}\} \tag{2}$$

$$x_{RMS_{worst}} = \max_{j \in [0..M-1]} \{x_{RMS_j}\} \tag{3}$$

In case we have a valid sorting but do not have its dependency graph, we can do the following: being $p \geq 0$ and $c \geq 0$ the total number of primitives and characters shown, respectively, we can compute the estimated RMS extreme values $\hat{x}_{RMS_{best}}$ and $\hat{x}_{RMS_{worst}}$ applying (1) as follows:

$$\hat{N}_{best} = p + 1 \tag{4}$$

$$\hat{N}_{worst} = \begin{cases} 1, \text{ for } p = 0 \\ p, \text{ for } p > 0 \end{cases} \tag{5}$$

$$\hat{x}_{RMS_{best}} = \sqrt{\frac{1}{p+1}\left(\frac{c}{p+1}\right)^2 (p+1)} = \frac{c}{p+1} \tag{6}$$

$$\hat{x}_{RMS_{worst}} = \begin{cases} c, \text{ for } p = 0 \\ \\ \sqrt{\frac{1}{p} \cdot c^2} = \frac{c}{\sqrt{p}}, \text{ for } p > 0 \end{cases} \tag{7}$$

Usually, the estimated RMS extreme values are unreachable, being $\hat{x}_{RMS_{best}} \leq x_{RMS_{best}}$ and $\hat{x}_{RMS_{worst}} \geq x_{RMS_{worst}}$. To get an idea, let's look at the real RMS value of the sorting made by Heisig in RTK1 (where $N = 166$, $c = 2200$, and $p = 165$):

$$x_{RMS} = \sqrt{\frac{1}{166} \cdot 81822} \approx 22.201$$

We don't know exactly the dependency graph he used, but comparing that value x_{RMS} with the estimated best and worst case...:

$$\hat{N}_{best} = 166$$
$$\hat{N}_{worst} = 165$$
$$\hat{x}_{RMS_{best}} = \frac{2200}{166} \approx 13.253$$
$$\hat{x}_{RMS_{worst}} = \frac{2200}{\sqrt{165}} \approx 171.269$$

...it seems that he did quite well to have done it manually; but in fact, that sorting can be improved to obtain a better learning because there are some intervals with zero characters and others quite long.

Finally, it should be noted that performance calculations should only be applied on valid sorted lists, because a sorting that violates Heisig's method may have better mathematical performance but not be a valid sorting. Also, we can only compare the performance of two valid sorted lists if they are generated from the same dependency graph... otherwise, the comparison would not make sense.

3 Implementation

The implementation of the algorithm has been developed as part of the investigation under *Myshella*, which is a personal project with a broad and idealistic objective: to bring order into chaos.

It has the following features:

- Multiset
  A character may belong to multiple sets. Some of the available sets are: Hiragana, Katakana, Jōyō kanji, Jinmeiyō kanji, Kyōiku kanji (by grade), Secondary school, Shinjitai, Kyūjitai, Kokuji, Hyōgaiji, JLPT (by level), 2500 most used kanji, RTK1, RTK3, etc.

- Modes
  It is able to generate sorted lists for any combination of supported sets. For instance, a sorted list for the union of Jōyō and Jinmeiyō kanji is perfectly possible.

- Edge labeling support
  In some situations, the use of edge labeling is helpful.

- Internal node types
  Heisig's method handles two types of nodes: primitives and characters. If we only used these types of nodes in the graph, the order obtained would be formally correct, but not entirely adequate for the student. This means that, in some cases, it is not enough to indicate the obvious dependencies. More types of nodes, which will not be shown in the sorted list, will be necessary to better structure the graph. The current implementation handles up to seven node types.

- Import criteria
  It exactly implements what seen in section 2.2: it handles three types of loading (Normal, Learned, and Imported), and a configurable parameter (the minimum number of child nodes that an imported character must have to be shown as a primitive).

- Sorting criteria
  These criteria are applied in order of priority, using the information available for the nodes: number of strokes, node type, edge label, number of dependencies, node weight in the graph... They are inspired by the criteria used by Heisig, although more complete.

- Joins
  There are a few cases where it is advisable to display some kanji consecutively because they always appear together in words (generally pairs of kanji, like in the Japanese word for *coffee* 珈琲). This implementation allows the option to make some exceptions to pure sorting, called *joins*, to automatically make appropriate modifications to the dependency graph and ensure that those kanji appear consecutively in the sorted list.

- Sorted list
  It gives either the first valid sorting found or the first among the best valid sorted lists, according to the proposed performance calculation seen in section 2.7.

4 Example

Let's see *Myshella* in action with a very little example: we have a list composed by nine characters (寸, 土, 下, 十, 圭, 二, 丁, 一, and 小) and three primitives (丨 / 亅 , 卜, and 丶). To create the dependency graph we have used four types of nodes (Figure 6).

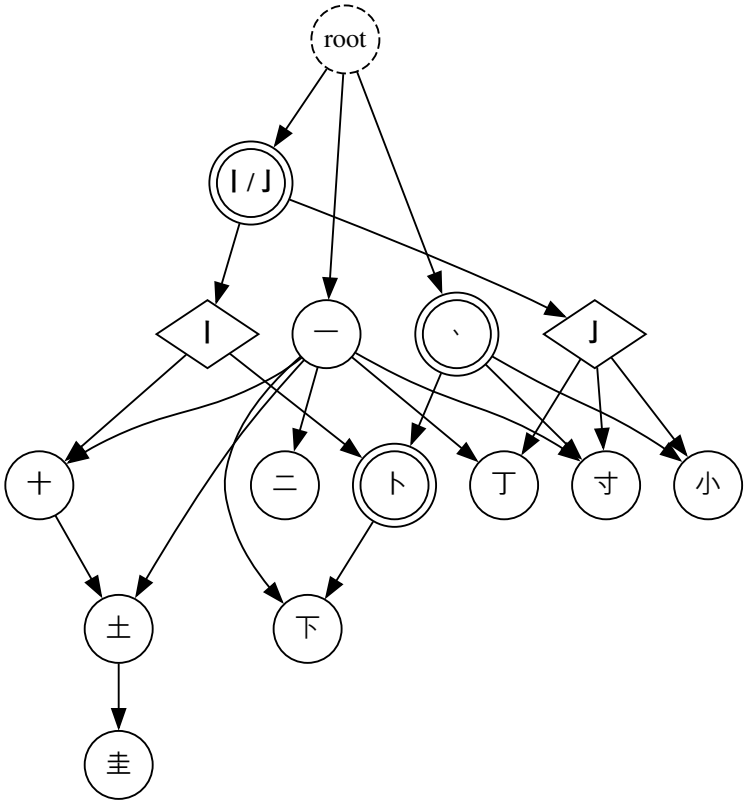


Figure 6: Example, using single circles for characters and double circles for primitives

Pay attention on primitive 丨 / 亅 , because it is an example of the subjectivity discussed in this text: here we have chosen to use a single primitive and divide it into two variants, but we could have done it differently; for instance:

- Only the primitive 丨 / 亅 , without the two variants (then, the dependencies for 寸 could be 十 and 丶).
- Two separated primitives, 丨 and 亅 , instead of primitive 丨 / 亅 .

Besides, primitive 丶 could be separated into two variants expressing two directions (both would be used in 小). In this example all this wouldn't be very important, but decisions like these in a graph with hundreds or thousands of nodes can potentially cause significant changes in the sorted list.

Well, applying *Myshella* will give us a valid sorting with performance $x_{RMS} = x_{RMS_{best}} = 2.5$, and the sorted list will be (primitives are indicated in lenticular brackets):

一 二 【丨/亅】 丁 十 土 圭 【丶】 小 寸 【卜】 下

As we see, we have $N = 4$ intervals and only characters and primitives are shown in the sorted list. In this simple example there is only one valid sorting using our selected performance calculation, so $x_{RMS_{best}} = x_{RMS_{worst}}$. Besides, if we only knew the sorted list but not the dependency graph, we could calculate the estimated extreme RMS values (being $c = 9$ and $p = 3$):

$$\begin{aligned} \hat{N}_{best} &= 4 \\ \hat{N}_{worst} &= 3 \\ \hat{x}_{RMS_{best}} &= \frac{9}{4} = 2.25 \\ \hat{x}_{RMS_{worst}} &= \frac{9}{\sqrt{3}} \approx 5.196 \end{aligned}$$

5 Conclusion

In the beginning, Heisig did a tremendous job creating his method and applying it to create sorted lists by hand, with the potential problems this entailed, but he did not explicitly formalize the method itself. Here, we have analyzed his method in order to present it formally and optimize its application in the following ways:

1. Applying it in a pure and systematic way to obtain better and easily adaptable sorted lists in response to changes in the character lists, making learning easier for the student and avoiding degradation in the sorting.
2. Selecting the best (or one of the best) sorting among all valid sorted lists thanks to the evaluation of performance.
3. In addition to improving Heisig's official sorted lists, we can apply his method to any other list of characters and primitives using the multiset support. For instance, it may be a good idea to create a sorted list for the entire RTK (3000 kanji), for the well-known list of 2500 most used kanji, or even for shorter kanji lists (by grade or level).

We must bear in mind that there is a degree of subjectivity when creating the model (data sets, import criteria, and sorting criteria) before running the method, but once this is assumed, we can objectively apply the proposed SP-DFS algorithm to obtain the best possible result.

References

[1] Heisig, James W. "Remembering the Kana: Remembering the Kana: A Guide to Reading and Writing the Japanese Syllabaries in 3 Hours Each". University of Hawai'i Press, 2007. ISBN 978-0-8248-3164-6.

[2] Heisig, James W. "Remembering the Kanji 1: A Complete Course on How Not to Forget the Meaning and Writing of Japanese Characters". University of Hawai'i Press, 2015. ISBN 978-0-8248-3592-7.

[3] Heisig, James W. "Remembering the Kanji 2: A Systematic Guide to Reading Japanese Characters". University of Hawai'i Press, 2012. ISBN 978-0-8248-3669-6.

[4] Heisig, James W. "Remembering the Kanji 3: Writing and Reading the Japanese Characters for Upper-Level Proficiency". University of Hawai'i Press, 2012. ISBN 978-0-8248-3702-0.

[5] Heisig, James W. "Remembering Simplified Hanzi 1: How Not to Forget the Meaning and Writing of Chinese Characters". Timothy W. Richardson. University of Hawai'i Press, 2009. ISBN 978-0-8248-3323-7.

[6] Heisig, James W. "Remembering Simplified Hanzi 2: How Not to Forget the Meaning and Writing of Chinese Characters". Timothy W. Richardson. University of Hawai'i Press, 2012. ISBN 978-0-8248-3655-9.

[7] Heisig, James W. "Remembering Traditional Hanzi 1: How Not to Forget the Meaning and Writing of Chinese Characters". Timothy W. Richardson. University of Hawai'i Press, 2009. ISBN 978-0-8248-3324-4.

[8] Heisig, James W. "Remembering Traditional Hanzi 2: How Not to Forget the Meaning and Writing of Chinese Characters". Timothy W. Richardson. University of Hawai'i Press, 2012. ISBN 978-0-8248-3656-6.

[9] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.