Compression Methods by Code Mapping and Code Dividing for Chinese Dictionary Stored in a Double-Array Trie

Huidan Liu
Institute of Software,
Chinese Academy of Sciences;
Graduate University of the
Chinese Academy of Sciences
huidan@iscas.ac.cn

Minghua Nuo
Institute of Software,
Chinese Academy of Sciences;
Graduate University of the
Chinese Academy of Sciences
minghua@iscas.ac.cn

Longlong Ma
Institute of Software,
Chinese Academy of Sciences
longlong@iscas.ac.cn

Jian Wu
Institute of Software,
Chinese Academy of Sciences
wujian@iscas.ac.cn

Yeping He
Institute of Software,
Chinese Academy of Sciences
yeping@iscas.ac.cn

Abstract

There is serious data sparseness problem in Chinese dictionary stored in a double-array trie. This paper proposes six compression methods by code mapping and code dividing to make it more compact, and a metric called Resource Consumption Ratio is proposed to evaluate these methods. Under the proposed criteria, five of the six methods are better than the baseline. The best method maps the character code into its frequency order, and then divides it into two jump codes. It achieves a space usage reduction of 39.88% and takes only 0.20% time of the baseline on the construction while it takes 13.21% more time on the retrieval. As preprocessing methods, these methods can be used to reduce more space by combining to other compression method which improves the double-array structure itself.

1 Introduction

In many applications of processing strings, a trie search is very useful because it enables fast retrieval and longest prefix matching with a small dictionary (Fredkin, 1960). Tries are used in a broad range of applications to represent a set of strings in fields such as information retrieval systems (Brain and Tharp, 1994; Nelson, 1997; Okada et al., 2001), lexical analyses (Aho et al., 2007; Lesk, 1975), morphological analyses (Aoe et al., 1996), natural language processing (Baeza-Yates and Gonnet, 1996; Peterson, 1980), bibliographic search (Aho et al., 1975), pattern matching (Flajolet and Puech, 1986), for IP address routing tables (Fu et al., 2007; Nilsson and Karlsson, 1999; Pao and Li, 2003), and text indexing (Navarro, 2004).

Double-array is a trie implementation which is proposed by (Aoe, 1989), and it is widely used in many applications at present because it combines the fast access of a matrix form with the compactness of a list form. However, there is a serious data sparseness problem when the double-array is used to store Chinese dictionary (Chinese double-array, hereafter, and similarly, English double-array for English dictionary). For a typical English dictionary with 45,373 words, the compression ratio of the double-array reaches 94.48%, but for a Chinese dictionary with 343,103 words, it's only 43.95%. In this paper, we analyze the reasons which lead to the data sparseness, and propose several methods to reduce the space usage of the Chinese double-array.

This paper is organized as follows: Section 2 introduces related work. Section 3 describes the outline of the double-array. In Section 4, we analyze the reasons resulting in the data sparseness. In Section 5, we give a baseline double-array, and propose six methods to make the double-array more compact. We propose the evaluation metric in Section 6. We make experiment and evaluate the six methods in Section 7. Section 8 concludes this paper.

2 Related work

Aoe (1989) presented an efficient digital search algorithm by introducing the structure called a double-array, which combines the fast access of a matrix form with the compactness of a list form.

Aoe (1992) proposed an implementation of the double-array which stores only as much of the prefix in the trie as is necessary to disambiguate the key, and the tail of the key is stored in a string array, denoted as TAIL. This structure is called a minimal prefix (MP) double-array.

Andersson and Nilsson (1993) introduced level-compressed (LC) trie to compress parts of the trie that are densely populated (Nilsson and Karlsson, 1999). A recent improvement in the LC trie is proposed by Fu et al. (2007).

Bentley and Sedgewick (1997) first presented the Ternary Search Tree (TST). TST combines the attributes of binary search trees and digital search tries.

Heinz (2002) proposed the burst trie, which is a collection of ”containers” that are accessed via a conventional trie. It achieves high space efficiency by selectively collapsing chains of trie nodes into small containers of strings that share a common prefix. Askitis and Sinha (2007) proposed an improvement of burst trie called the HAT-trie which uses cache-conscious hash tables. Compared with a TST, the burst trie is 25% faster and uses only 50% of the memory, though it was found to be slower than TST with genomic data (Heinz et al., 2002).

Oono (2003) presented a method of dividing a key into several parts and defining link information between keys. It turned out that the double-array is 30% smaller than old method.

Wang (2006) proposed an improved strategy for the double-array. The node with most child nodes is inserted firstly while constructing, which reduces the data sparseness and keeps the search efficiency. Li (2006) designed and implemented a Chinese dictionary based on the double-array. Wang et al. (2009) introduced the idea of Sherwood random thoughts and mutation of genetic algorithms to improve the performance of the method proposed by Wang (2006) to avoid catching the trap of local optimal solution.

Yata (2007) presented two compaction methods for a static MP double-array, an element compaction and a trie compaction. The element compaction reduces the size of each element. The trie compaction reduces the number of nodes (a descended trie) and the length of the array keeping suffixes. The space usage for the new double-array is under the half of that for the original one, but the compaction methods little degrade the retrieval speed of the double-array.

Dorji (2010) presented three methods to compress the MP double-array. The first two methods accommodate short suffixes inside the leaf nodes, and prune leaf nodes corresponding to the end marker symbol. They achieve size reduction of up to 20%, making insertion and deletion faster at the same time while keeping the retrieval time of $O(1)$. The third method eliminates empty spaces in the array that holds suffixes, and improves the size reduction further by about 5% at the cost of increased insertion time. Compared to a TST, the key retrieval of the compressed double-array is 50% faster and its size is 3-5 times smaller.

All the compression methods are focusing on the corresponding trie or the double-array structure itself. However, the space usage of a double-array is very relevant to the content which is stored in it. We will compress the double-array by preprocessing the content.

3 Outline of double-array

A double-array is an array form of a trie (Aoe, 1989). It uses two one-dimensional arrays, named BASE and CHECK, to represent a trie. An element of the trie consists of the two array units with the same index in BASE and CHECK. Every element corresponds to a node of the trie, except empty elements, and we will also take the nonempty element as a node in this paper. For a certain node, the unit in BASE indicates the offset to child nodes, while the unit in CHECK normally stores the index of the parent node.

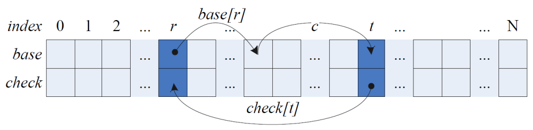


Figure 1: Relation between a node t and its parent node r in the double-array

As shown in Figure 1, in a double-array, a node t is a child of another node r, if and only if there is a relation between the two nodes.

$$\begin{cases} BASE[r] + c & = t \\ CHECK[t] & = r \end{cases} \tag{1}$$

where c is a numerical value corresponding to a character in a key, which we call ”jump code” because it leads to a jump (state transition) from a node to its child node.

The retrieval of a key is just a walk from the root node to a leaf node, which is done fast by array operations in the double-array. The time complexity for a state transition from node r to its child node t is $O(1)$ in the double-array, thus the time complexity for retrieving a key is $O(k)$ for the length k of that key.

4 Data sparseness problem analysis

Unlike alphabetic writing script, Chinese is an ideographic script which has a large character set. There are more than 70,000 coded Chinese characters in Unicode 6.0. Although most of them are historic characters, there are still a large amount of characters that are frequently used. As in the Chinese dictionary mentioned above, there are more than 14,000 characters used.

The data sparseness problem of Chinese double-array mainly results from the large character set. First, jump codes vary in a large interval. In Unicode 6.0, the code of Chinese character varies from 3400 to 9FCB, and there are still a large amount of characters out of the range because they are out of the basic multilingual plane and should be represented by surrogate pairs which are in the interval from D800 to DFFF. But, not all of the characters are used in the dictionary. As a result, there are many empty elements between the most left child node and the most right one from the same parent in the double-array while inserting, although some of them may be used to store other nodes soon after. Second, a node has much more child nodes in Chinese trie than in English. In the Chinese dictionary, there are more than 800 words which begin with the same character ”” meaning ”one”, which indicates that the corresponding node will have more than 800 children. However, in an English trie, the number of child nodes for any node doesn’t exceed 52 for there are only 26 letters in the alphabet. More child nodes lead to more collision, which results in more empty elements indirectly.

5 Compression of Chinese double-array

In this section, we first present a baseline double-array, and then propose six methods of compaction. For each method, we only focus on the space usage, and compare it with the baseline from node count (NC), array length (AL), auxiliary space and compression ratio (CR). CR is the ratio of the number of non-empty elements to the total number of elements in the double-array. It represents the compactness of the double-array. We will use it to estimate how much room there is to apply compression method to reduce the space usage. We evaluate each method by calculating the space reduction rate (SRR), which is the ratio of the total space usage of each method to that of the baseline.

5.1 Baseline

Simply, we take the double-array proposed by (Aoe, 1989) which uses directly the code of a character as the as the baseline. Generally, there is an array called TAIL (Aoe et al., 1992) which stores suffix strings in the MP double-array. But we use the original double-array which store all the whole keys rather than only the minimal prefixes as the baseline to compute the space usage conveniently. As each unit in the array is an integer which takes up 4 bytes memory, the total space usage can be calculated as follow: $8bytes \times len$, where len is the length of the double-array.

We use the character code as the jump code in the baseline. For the key set K={“中国”, “中国象棋”, “中间”, “上海”, “上浮”} in which it means “China”, “Chinese chess”, “middle”, “Shanghai” (a city of China), ”floating upward” respectively, the corresponding trie is as shown in Figure 2. Note that a node with two concentric circles corresponds to an acceptance state which indicates the end of a word.

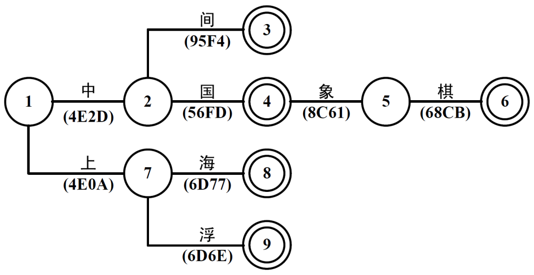


Figure 2: The trie of the key set K.

Based on the trie shown in Figure 2, a Chinese double-array is built to store the Chinese dictionary. As it will be represented by a surrogate pair if a character is out of the basic multilingual plane, we will take it as two characters for convenience. The baseline takes 1,714,339 units for the BASE and CHECK respectively, in which only 753,412 units are not empty, and the CR is 43.95%. As we analyzed in the former section, the first method we can think out is mapping the jump codes into a small interval.

5.2 Code mapping methods

As shown in Figure 3, this method maps the codes of all characters used in the dictionary into a s-mall continuous interval. BASE stores the offsets from each node to its child nodes, and differen-t start points only lead to different offsets, which hardly affect the space usage. So the start point of the interval is not very important. In this paper, we take 80 (hex) as the start point to make it compatible with English script. The mapping just changes the value of each jump code, while the trie keeps unchanged.

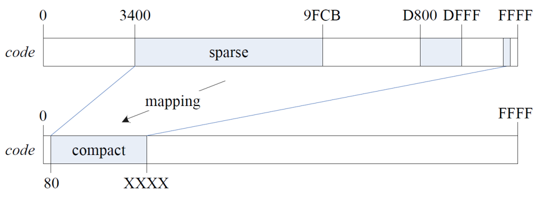


Figure 3: Code mapping.

We have to use another array to store the mapping table. Each unit takes 2 bytes, and 65536 (10000 in hex) units are needed to include al-l codes in basic multilingual plane. The auxiliary space is: 2 $bytes \times 65536 = 131072$ $bytes$.

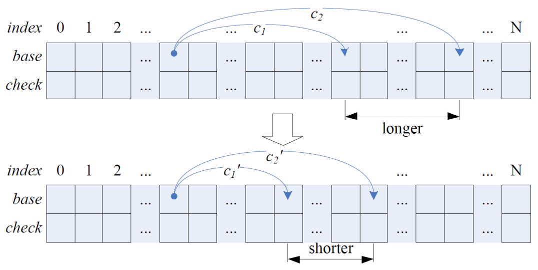


Figure 4: Double-arrays before and after applying code mapping method.

As shown intuitively in Figure 4, before applying code mapping method, for every node in the trie, there is a long distance between the most left child node and the most right one. After apply the method, the distance becomes much shorter. The number of child nodes keeps the same, data is more compact for the parent node, and the collision is controlled in a small range in the double-array while inserting other nodes. As child nodes for every intermediate node are arranged more compactly, it alleviates the data sparseness problem more or less.

Based on code mapping, we introduce the following two methods:

Method 1: Mapping each character code to it-s original order number in Unicode. The method takes the order number as the jump code for each character used in the dictionary, and keeps their original codes for others during the retrieval pro-cess. The relation between the target jump code c' and the original jump code c can be represented by the following formula:

$$c' = order(c) + C \tag{2}$$

where $order(c)$ is the order number of c among all the Chinese characters used in the dictionary when they are sorted in ascending order by code, and C is a constant which indicates the start point of the target interval. We set C equal to 80 (hex).

This method is already used by many people (Aoe, 1989; Aoe et al., 1992; Wang et al., 2006; Li et al., 2006; Dorji et al., 2010). Wang (2006) and Li (2006) used this method in their dictionary, but they just simply map the Chinese characters coded in the Chinese standard GB 2312 to 1∼6763.

As shown in Table 1, after the mapping, the CR is improved to 47.49%, and it reduces the space usage of the double-array by 6.51% in total con-sidering the auxiliary space. As the CR is still very low, there is much room for improvement.

	NC	AL	AS (bytes)	CR	SRR
Baseline	753412	1714339	0	43.95%	
Method 1	753412	1586337	131072	47.49%	6.51%

Table 1: Space usage of Method 1 compared with the baseline.

In Method 1, we make code mapping according to their original order in Unicode. However, different characters have different frequencies in the dictionary, then what will happen if we map two characters with the same frequency to two target jump codes of which the numerical difference is as small as possible?

Method 2: Mapping the jump codes to their frequency orders. Then, all high frequency characters will be mapped into a small interval. If two characters leading to state transitions from the same n-ode to its two child nodes are both frequency used in the dictionary, then the distance between the t-wo child nodes will be shorter than in Method 1. The relation between the target jump code c' and the original one c is represented by the following formula:

$$c' = freqorder(c) + C \tag{3}$$

where $freqorder(c)$ is the order number of c a-mong all the Chinese characters used in the dictionary when they are sorted by frequency in descending order.

As shown in Table 2, after the mapping, the CR is improved to 58.56%, and it reduces the space usage of the double-array by 23.99% in total.

	NC	AL	AS (bytes)	CR	SRR
Baseline	753412	1714339	0	43.95%	
Method 2	753412	1286622	131072	58.56%	23.99%

Table 2: Space usage of Method 2 compared with the baseline.

Compared with Method 1, Method 2 makes a significant improvement, but there is still room for improvement.

5.3 Code dividing methods

We make all jump codes varying in a small interval by code mapping, but the interval is still much larger than that in the English double-array. It spreads from 80 (hex) to 3730 (hex), for there are more than 14,000 Chinese characters are used in the dictionary. However, in a English double-array, the jump codes vary in a much smaller interval, which is from 41 (hex of letter "A") to 7A (hex of letter "z") originally. To compress the interval equivalent to that of English, we divide each original jump code into two or more jump codes to break a long jump in the original double-array into two much shorter jumps.

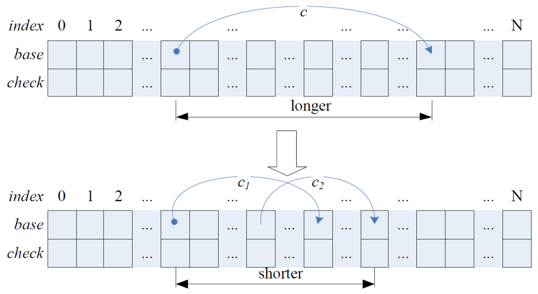


Figure 5: Double-arrays before and after applying code dividing methods.

As shown in Figure 5, every original jump code is divided into two jump codes. A state transition in the original double-array becomes two state transitions in the new double-array. The node (state) count increases. But it becomes more compact, it's still possible to reduce the space usage.

Method 3: Divide the code of a Chinese character into two small codes. To ensure that the two codes vary in equivalent interval, we take the high byte of the code as the first target code, and the low byte as the second. To make it compatible with English, we also change the start point of the interval to 80 (hex) as we do in Method 1. The relation between the two target jump codes c1, c2 and the original jump code c can be represented by Formula 4.Note that in the formula, $(c \gg 8)$ and $(c\&FF)$ are just the high/low byte of the c.

$$\begin{cases} c1 &= (c \gg 8) &+ C \\ c2 &= (c \,\&\, FF) &+ C \end{cases} \tag{4}$$

Generally, the sum of c1 and c2 is much smaller than c. For example, for a certain Chinese character "一", $c = 4E00$, then $c1 = CE(hex), c2 = 80(hex)$, and $c1 + c2 = 014E(hex)$, which is much smaller than c1. It results in shorter distance between a parent node and its child nodes, and shorter distance between different child nodes from the same parent.

As every original jump code is divided into two jump codes, the corresponding trie changes. For the key set K, the corresponding trie is shown in Figure 6. Although each jump code is divided into two, the total node count is smaller than the twice of the original trie's, because some nodes are shared, just like the nodes 2 in Figure 6 because "中" and "上" have the same high byte. It's similar to node 13.

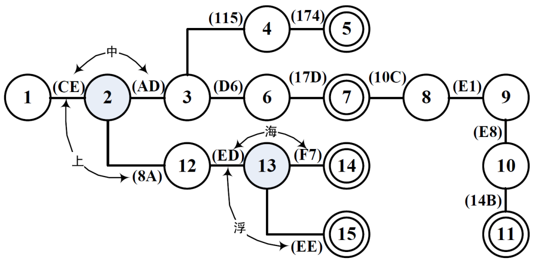


Figure 6: The new trie of key set K.

	NC	AL	AS (bytes)	CR	SRR
Baseline	753412	1714339	0	43.95%	
Method 3	998995	1043342	0	95.75%	39.14%

Table 3: Space usage of Method 3 compared with the baseline.

As shown in Table 3,every original code is divided into two codes, the node count of the double-array increases by 32.60% from 753412, but the CR is improved to 95.75%, which is similar to English double-array. So the length of the double-array is still smaller than the baseline. Meanwhile, Method 3 doesn't use a mapping table. It needs no auxiliary space. Thus, the space reducing rate reaches 39.14% in total.

Then what will happen if we divide the original code into 3 or more codes?

Method 4: Take the UTF-8 sequence of Chinese character as the corresponding jump codes.

	NC	AL	AS (bytes)	CR	SRR
Baseline	753412	1714339	0	43.95%	
Method 3	998995	1043342	0	95.75%	39.14%
Method 4	1200403	1264949	0	94.90%	26.21%

Table 4: Space usage of Method 3 and Method 4 compared with the baseline.

As shown in Table 4, after applying Method 4, the node count increases nearly 60% compared with the baseline, which leads to more space usage than Method 3. The CR of Method 4 is a little smaller than that of Method 3. The space usage reduction rate is only 26.21%, which is much smaller than Method 3. In a word, Method 3 is better than Method 4.

5.4 Combined methods

Then, what will happen if we combine methods of code mapping and code dividing? Let's have a try. Combining Method 3 to each of the code mapping methods, we get Method 5 and Method 6. Method 4 is worse than Method 3. We won't try to combine it to the code mapping methods.

Method 5 (Method 1 + Method 3): First, map the original character to its order number, and then divide it into two target jump codes. The corresponding formula of Method 5 is as follow:

$$\begin{cases} c1 & = (order(c) \gg 7) & + C \\ c2 & = (order(c) \,\&\, 7F) & + C \end{cases} \tag{5}$$

Method 6 (Method 2 + Method 3): First, map the original character to its frequency order number, and then divide it into two jump codes. The corresponding formula of Method 6 is as follow:

$$\begin{cases} c1 & = (freqorder(c) \gg 7) & + C \\ c2 & = (freqorder(c) \,\&\, 7F) & + C \end{cases} \tag{6}$$

Note that we don't shift by 8 bits any longer like in Method 3 and Method 4, because the total number of characters used in the dictionary is about 14 thousand, and so it needs only 14 valid bits to represent all the order numbers.

As shown in Table 5, Method 5 and Method 6 both need auxiliary space to store the mapping table like in Method 1 and Method 2. They both

	NC	AL	AS (bytes)	CR	SRR
Baseline	753412	1714339	0	43.95%	
Method 5	972703	1014204	131072	95.91%	39.88%
Method 6	925643	962857	131072	96.14%	42.88%

Table 5: Space usage of Method 5 and Method 6 compared with the baseline.

achieve a CR larger than 95% and reduce the space usage by a percentage near or even larger than 40%, which is similar to Method 3. However, Method 6 is slightly better than Method 5 from the aspect of space usage, just like Method 2 is slightly better than Method 1.

Now, we don't think there is still much room for improvement because the CR is close to 100% and neither combined method makes big improvement compared with Method 3.

6 Evaluation metric

If we evaluate several methods or systems from n aspects of resources consumption such as space usage and time cost, and then we denote the value on the ith aspect by V_{Ai} and the normalized weight of the ith aspect by w_i. We define the resource consumption ratio (RCR) of two methods A and B as follow:

$$RCR(A, B) = \prod_{i=1}^{n} \left(\frac{V_{Ai}}{V_{Bi}} \right)^{w_i} \tag{7}$$

RCR has the following properties:

- $RCR(A, A) = 1.0$, if a method A is compared with itself;
- $RCR(A, B) = 1.0$ means the two methods A and B are equally good or bad;
- $RCR(A, B) > 1.0$ means the method A is worse than the reference method B, because it consumes more resources.
- $RCR(A, B) < 1.0$ means the method A is better than the reference method B because it consumes fewer resources.

7 Experiment and Evaluation

In this section, We evaluate the six methods from the following aspects:

1. Total space usage (TSU), including the space for the double-array and the mapping table.
2. Construction time (CT), including the time spent on making the code mapping or the code dividing.

3. Retrieval time (RT) on searching all words in the dictionary.

7.1 Experiment

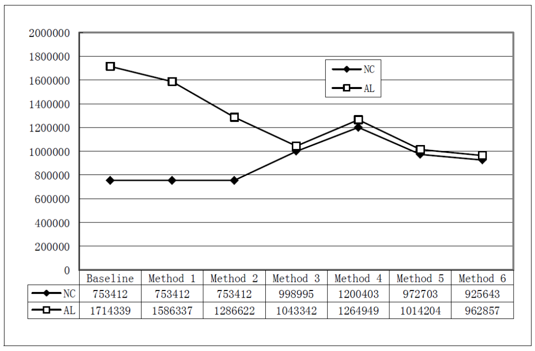
The dictionary used in this paper is the Modern Chinese Dictionary in Traditional Chinese.

Normally, a double-array allocates space many times to extend space to accommodate newly inserted nodes during the construction. As different methods have different space usage, they may need different times of space allocation, and thus need different time cost. We simply allocate a large enough space in advance for the double-array before the construction to eliminate the difference of time cost on space allocation.

It takes a short time to retrieve all words in the dictionary once. If we execute the experiment twice, the time costs will be very different from each other if it is interfered by some accidental factors. We do the retrieval 1000 times to alleviate the problem mostly and keep the time costs comparable.

The experiment is performed on a computer with an 8-core 2.80GHz Intel Core i7 CPU and 2.96GB memory. The experimental data is shown in Table 6. Note that SUR, CTR, RTR is the ratio of TSU, CT, RT in each method to that in the baseline respectively.

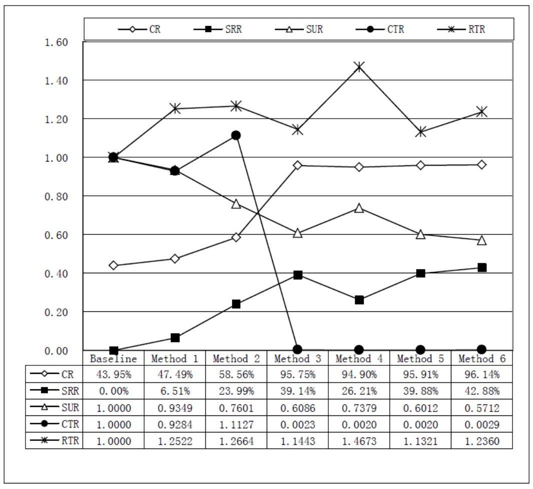
7.2 Evaluation



	Baseline	Method 1	Method 2	Method 3	Method 4	Method 5	Method 6
NC	753412	753412	753412	998995	1200403	972703	925643
AL	1714339	1586337	1286622	1043342	1264949	1014204	962857

Figure 7: Node count and array length.

As we see from Figure 7 and Figure 8, Method 1 and Method 2 achieve space usage reduction of 6.51% and 23.99% respectively, but they take too much time on the construction. Method 3 achieves a space usage reduction of 39.14% while it takes very short time on the construction. It takes a little longer time on the retrieval than the baseline. Method 4 also takes very short time on the construction, but much more time on the retrieval and it only achieves a space usage reduction of 26.21%. Method 5 achieves a better space usage reduction, short time cost both on construction and retrieval than Method 3. Method 6 achieves the best space usage reduction by 42.88%, but it takes longer time cost on the construction and retrieval than Method 5. Method 3∼6 have more nodes because they have divided original codes into two or more codes, but they makes the double-array more compact, so every one of them has a CR close to 97%. Method 6 takes more time on the construction than Method 5, which is similar to Method 2 and Method 1, because it has to spend time to sort the characters by frequency. All the methods take more retrieval time, but result from different reasons. Method 1 and Method 2 have to make the code mapping while Method 3∼6 have to make the code dividing and visit more nodes while searching a key.



	Baseline	Method 1	Method 2	Method 3	Method 4	Method 5	Method 6
CR	43.95%	47.49%	58.56%	95.75%	94.90%	95.91%	96.14%
SRR	0.00%	6.51%	23.99%	39.14%	26.21%	39.88%	42.88%
SUR	1.0000	0.9349	0.7601	0.6086	0.7379	0.6012	0.5712
CTR	1.0000	0.9284	1.1127	0.0023	0.0020	0.0020	0.0029
RTR	1.0000	1.2522	1.2664	1.1443	1.4673	1.1321	1.2360

Figure 8: Comparison of the six methods.

Now, we evaluate the six methods with the metric proposed in the former section from three aspects: space usage, construction time, retrieval time. We assign the three aspects with weights (9/20, 1/10, 9/20) respectively. The construction is only performed once but the retrieval is performed many times for a double-array, so the construction time is assigned a smaller weight, and the space usage has an equal weight to retrieval time because we can't decide which is more important roughly without any application scenarios. We compare each method A with the baseline to calculate each RCR:

$$\begin{aligned} RCR(A) &= RCR(A, baseline) \\ &= SUR^{\frac{9}{20}} \times CTR^{\frac{1}{10}} \times RTR^{\frac{9}{20}} \end{aligned} \tag{8}$$

We calculate the RCR of each method with the

	NC	AL	CR	SRR	TSU(byte)	SUR	CT(ms)	CTR	RT(ms)	RTR	RCR
Baseline	753412	1714339	43.95%	0.00%	13714712	1.0000	504437	1.0000	23172	1.0000	1.0000
Method 1	753412	1586337	47.49%	6.51%	12821768	0.9349	468328	0.9284	29015	1.2522	1.0655
Method 2	753412	1286622	58.56%	23.99%	10424048	0.7601	561265	1.1127	29344	1.2664	0.9935
Method 3	998995	1043342	95.75%	39.14%	8346736	0.6086	1140	0.0023	26516	1.1443	0.4621
Method 4	1200403	1264949	94.90%	26.21%	10119592	0.7379	1015	0.0020	34000	1.4673	0.5570
Method 5	972703	1014204	95.91%	39.88%	8244704	0.6012	984	0.0020	26234	1.1321	0.4506
Method 6	925643	962857	96.14%	42.88%	7833928	0.5712	1484	0.0029	28641	1.2360	0.4773

Table 6: Comparison of the six methods and the baseline.

values collected from the experiment, and the results are shown in Figure 9.

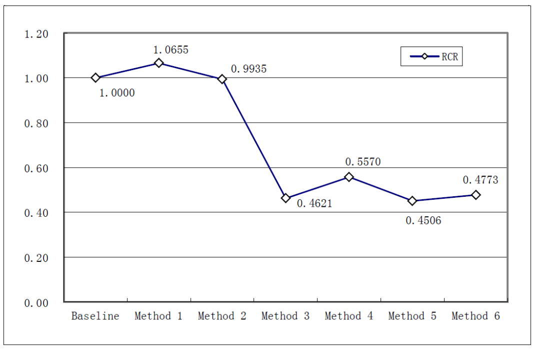


Figure 9: RCRs of the six methods.

As we see in Figure 9, most of the six methods are better than the baseline except Method 1, whose RCR is slightly larger than 1.0. Method 5 has the smallest RCR and it outperforms others.

8 Conclusion

We propose six methods to reduce the space usage of Chinese double-array. Method 1 and Method 2 achieve space usage reduction of 6.51% and 23.99% by code mapping into the order number or frequency order number respectively, but they take too much time on the construction. Method 3 achieves a space usage reduction of 39.14% by dividing each original jump code into two codes while it takes very short time on the construction. It takes a little longer time on the retrieval than the baseline. Method 4 divides the original jump code into three codes. It also takes very short time on the construction, but much more time on the retrieval and it only achieves a space usage reduction of 26.21%. Method 5 and Method 6 are the combinations of Method 3 with Method 1 and Method 2 respectively. Method 5 achieves a better space usage reduction, short time cost both on construction and retrieval than Method 3. Method 6 achieves the best space usage reduction by 42.88%, but it takes longer time cost on the construction and retrieval than Method 5.

Also, an approach is proposed to evaluate those methods by a metric called Resource Consumption Ratio (RCR) which compares the total resource consumption of two methods such as space usage and time cost. Under the proposed criteria, five of the six methods are better than the baseline, and the best one (Method 5) achieves a space usage reduction by 39.88% and takes only 0.20% time of the baseline on the construction, while it takes 13.21% more time on the retrieval. As preprocessing methods, these methods can be used to reduce more space by combining to other compression method which improves the double-array structure itself.

Acknowledgements

We would like to thank the anonymous reviewers for their critical and constructive comments and suggestions that helped us improve the quality of the paper. The research is partially supported by National Science and Technology Major Project (No.2010ZX01036-001-002, No.2010ZX01036-001-002), and CAS Action Plan for the Development of Western China (No.KGCX2-YW-512).

References

Alfred V. Aho, and Margaret J. Corasick. 1975. *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, 18(6):333–340.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers principle tech-niques and tools (2nd Edition)*. Addison-Wesley.

Arne Andersson, and Stefan Nilsson. 1993. *Efficient implementation of suffix trees*. Software: Practice and Experience, 25(2):129–141.

Jun-ichi Aoe, Katsushi Morimoto, and Takashi Sato. 1992. *An efficient implementation of trie structures*. Software-Practice and Experience, 22(9):695–721.

Jun-ichi Aoe, Katsushi Morimoto, Masami Shishibori, and Ki-Hong Park. 1996. *A trie compaction algorithm for a large set of keys*. IEEE Transactions on Knowledge and Data Engineering, 8(3): 476–491.

Jun-ichi Aoe. 1989. *An efficient digital search algorithm by using a double-array structure*. IEEE Transactions on Software Engineering, 15(9): 1066–1077.

Nikolas Askitis, and Ranjan Sinha. 2007. *HAT-trie: A cache-conscious trie-based data structure for strings*. In Proceedings of the 30th Australasian computer science conference, Ballarat, Australia.

Ricardo A. Baeza-Yates, and Gaston H. Gonnet. 1996. *Fast text searching for regular expressions or automaton searching on tries*. Journal of the ACM, 43(6):915–936.

Jon L. Bentley, and Robert Sedgewick. 1997. *Fast algorithms for sorting and searching strings*. In Proceedings of the 8th annual ACM-SIAM sympo-sium on discrete algorithms.

Marshall D. Brain, and Alan L. Tharp. 1994. *Using tries to eliminate pattern collisions in perfect hashing*. IEEE Transactions on Knowledge and Data Engineering, 6(2):239-247.

O'Neil Delpratt, Naila Rahman, and Rajeev Raman. 2006. *Engineering the LOUDS succinct tree representation*. In fifth international workshop on experimental algorithms, WEA2006. LNCS 4007: 134–145.

Masaki Dono, Masako Fuketa, Yutaka Inada, Yo Murakami, and Jun-ichi Aoe. 2006. *A Compression Method Using Link-Trie Structure for Natural Language Dictionaries*. International Conference on Computing and Informatics. 1–4.

Tshering C. Dorji, El-sayed Atlam, Susumu Yata, Mahmoud Rokaya, Masao Fuketa, Kazuhiro Morita, and Jun-ichi Aoe. 2010. *New methods for compression of MP double array by compact management of suffixes*. Information Processing and Management, 46:502–513.

John A. Dundas. 1991. *Implementing dynamic minimal prefix tries*. Software-Practice and Experience, 21(10):1024-1040.

Philippe Flajolet, and Claude Puech. 1986. *Partial match retrieval of multidimensional data*. Journal of the ACM, 33(2):371–407.

Philippe Flajolet. 2006. *The ubiquitous digital tree. In Proceedings in lecture notes in computer science*. LNCS 3884:1–22. Berlin/Heidelberg: Springer.

Edward Fredkin. 1960. *Trie memory*. Communications of the ACM, 3(9):490–500.

Jing Fu, Olof Hagsand, and Gunnar Karlsson. 2007. *Improving and analyzing LC-Trie performance for IP-address lookup*. Journal of Networks, 2:18–27.

Steven Heinz, Justin Zobel, and Hugh E. Williams. 2002. *Burst tries: A fast, efficient data structure for string keys*. ACM Transactions on Information Systems, 20(2):192–223.

Lesk, M. E. (1975). *Lex - a lexical analyzer generator*. CSTR 39. NJ: Bell Lab, 1–13.

Jiangbo Li, Qiang Zhou, and Zushun Chen. 2006. *A Study on Fast Algorithm for Chinese Dictionary Lookup*. Journal of Chinese Information Processing, 20(5):31–40.

Gonzalo Navarro. 2004. *Indexing text using the Ziv-Lempel trie*. Journal of Discrete Algorithms, 2(1): 87–114.

Michael J. Nelson. 1997. *A prefix trie index for inverted files*. Information Processing and Manage-ment, 33(6):739–744.

Stefan Nilsson and Gunnar Karlsson. 1999. *IP-address lookup using LC-tries*. IEEE Journal on Selected Areas in Com-munication, 17(6):1083–1092.

Makoto Okada, Kazuaki Ando, Sangkon, Yoshitaka Hayashi, and Jun-ichi Aoe. 2001. *An efficient substring search method by using delayed keyword extraction*. Information Processing and Management, 37:741-761.

Masaki Oono, El-Sayed Atlam, Masao Fuketa, Kazuhiro Morita, and Jun-ichi Aoe. 2003. *A fast and compact elimination method of empty elements from a double-array structure*. Software-Practice and Experience, 33, 1229–1249.

Derek Pao and Yiu-Keung Li. 2003. *Enabling incremental updates to LC-trie for efficient management of IP forwarding tables*. Communications Letters, IEEE, 7(5), 245–247.

James L. Peterson. 1980. *Computer programs for spelling correction: An experiment in program design*. LNCS 96:1–129. Springer-Verlag.

Guy Shani, Christopher Meek, Tim Paek, Bo Thiesson, and Gina Venolia. 2009. *Searching large in-dexes on tiny devices: Optimizing binary search with character pinning*. In Proceedings of the 13th international conference on Intelligent user interfaces, 257–266.

Sili Wang, Huaping Zhang, and Bin Wang. 2006. *Research of Optimization on Double-Array Trie and its Application*. Journal of Chinese Informa-tion Processing. 20(5): 24–31.

Shikun Wang, Shaozi Li, and Xiao Ke. 2009. *Double-array Trie based on genetic algorithm and idea of Sherwood*. Computer Engineering and Application-s, 45(29):128–130.

Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun-ichi Aoe. 2007. *A compact static double-array keeping character codes Source*. Information Processing and Management, 43(1): 237–247.