# Experimental Observations of Construction Methods for Double Array Structures Using Linear Functions

Shunsuke Kanda*, Kazuhiro Morita, Masao Fuketa, Jun-Ichi Aoe
Department of Information Science and Intelligent Systems, Faculty Engineering, University of Tokushima, Tokushima, Japan

* Corresponding author. Email: c501437039@tokushima-u.ac.jp.
Manuscript submitted August 1, 2014; accepted March 8, 2015.
doi: 10.17706/jsw.10.6.739-747

**Abstract:** A trie is an ordered tree data structure to store keywords. It is used in natural language processing and so on. The trie is represented by the double array. The double array can retrieve fast at time complexity of $O(1)$. The double array using linear functions (DALF) is proposed as a compression method of the double array. DALF reduces space usage of the double array to 60%. DALF is built by using parameters, and its space usage and its construction time depend on these parameters. However, appropriate values of them are not determined. This paper observes these parameters and evaluates parameters by experiments. From experiments, appropriate parameters are found, and it turns out that DALF can be built more efficiently by keyword sets including multibyte characters.

**Key words:** Trie, double array, construction method, keyword search.

## 1. Introduction

A trie [1] is a tree structure for keyword retrieval. In the trie, each keyword is registered as the path from the root node to the leaf node, and the prefixes of keywords are merged. Therefore, the trie can retrieve the longest prefixes fast. Because of this merit, the trie is utilized in natural language processing [2], searching for reserved words for compilers [3], IP address lookup [4], and so on [5], [6].

The double array presented by Aoe [7] is an efficient data structure that represents the trie with two one-dimensional arrays called BASE and CHECK. The double array provides fast retrieval at time complexity of O(1). BASE and CHECK are arrays of signed integers and have the same space usage. Each element of BASE and CHECK need 4 bytes and 4 bytes, respectively. Therefore, the space usage of the double array is 8|D| bytes (|D| is the number of the double array's elements).

To reduce the space usage of the double array, Yata *et al*. presented a compacted double array (CDA) [8] and Fuketa *et al.* presented a single array with multi code (SAMC) [9]. CDA is a method that CHECK keeps character codes. Each element of CHECK needs 1 byte, and the space usage of CDA is 5|D| bytes. SAMC is a method that BASE is deleted and CHECK keeps character codes. Its space usage is |D| bytes. However, |D| increases depending on sets of keywords.

The double array using linear functions (DALF) presented by Kanda *et al*. [10] is a more compact method for the double array. DALF reduces each element of BASE to 2 bytes by using linear functions, and then the space usage of DALF is 3|D| bytes. DALF reduces space usage of CDA to 60%. In construction algorithms, DALF uses two parameters gain and a. When these parameters are appropriate, its space usage becomes compact and its construction time becomes short. However, in [10], because it is difficult to choose appropriate them, definitions of them are not written clearly.

In this paper, DALF is built by using various combinations of parameters gain and a, and is evaluated

## 2. Double Array

### 2.1. Outline of the Double Array

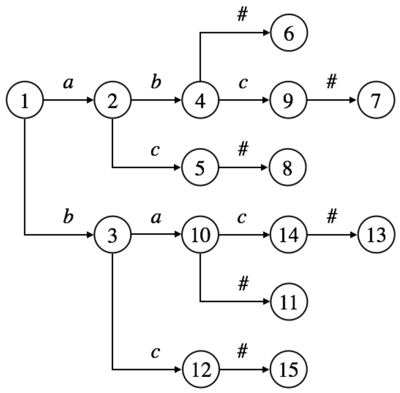
The double array uses two one-dimensional arrays called BASE and CHECK in order to represent trie nodes. For example, element *s* of the double array consists of BASE[s] and CHECK[s] corresponding to node *s* in the trie. The following equations show an arc from node *s* to node *t* with character *c*;

$$\mathrm{BASE}[s]+\mathrm{CODE}[c]=t;\mathrm{CHECK}[t]=s \tag{1}$$

The index of destination node *t* is calculated by the sum of the offset BASE[*s*] and CODE[*c*] that is the numerical code of character *c*. The index of source node *s* is stored in CHECK[*t*]. Each element of BASE and CHECK respectively require 4 bytes and 4 bytes because these store integer values. The space usage of the double array is 8|*D*| bytes. Fig. 1 shows a double array of keyword set *K* = {"*ab*", "*abc*", "*ac*", "*ba*", "*bac*", "*bc*"}. Special end marker '*#*' is used at the end of keys.

### 2.2. Outline of the Compacted Double Array

A compacted double array (CDA) reduces the space usage of the double array. In CDA, (1) is changed as follows;




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE | 1 | 2 | 9 | 6 | 8 | | | | 7 | 11 | | 15 | | 13 | |
| CHECK | | 1 | 1 | 2 | 2 | 4 | 9 | 5 | 4 | 3 | 10 | 3 | 14 | 10 | 12 |

| | # | a | b | c |
|---|---|---|---|---|
| CODE | 0 | 1 | 2 | 3 |

Fig. 1. The double array for *K*.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BASE | 1 | 2 | 9 | 6 | 8 | | | | 7 | 11 | | 15 | | 13 | |
| CHECK | | *a* | *b* | *b* | *c* | # | # | # | *c* | *a* | # | *c* | # | *c* | # |

| | # | a | b | c |
|---|---|---|---|---|
| CODE | 0 | 1 | 2 | 3 |

Fig. 2. The compacted double array for *K*.

$$\mathrm{BASE}[s]+\mathrm{CODE}[c]=t;\mathrm{CHECK}[t]=c. \tag{2}$$

Simultaneously, the following equation requires to be satisfied in all pairs of nodes {*i*, *j*} except the leaf nodes.

$$\mathrm{BASE}[i]\neq \mathrm{BASE}[j] \tag{3}$$

A character is stored in CHECK. Each element of CHECK requires 1 byte and the space usage of the double array is 5|*D*| bytes. Fig. 2 shows CDA for key set *K*.

## 3. Double Array Using Linear Function

### 3.1. Outline of the Double Array Using Linear Functions

The double array using linear functions (DALF) compresses the space usage of CDA. DALF divides the trie into each depth and defines linear functions $fd(s)$ for each depth ($d$ (≥1) is the depth of the trie). Equation (2) is changed for DALF as follows;

$$\mathrm{DBASE}[s] + f_d(s) + \mathrm{CODE}[c] = t; \mathrm{CHECK}[t] = c. \quad (4)$$

Each element of DBASE needs 2 bytes. $f_d(s)$ is the linear function with index $s$ of the double array, and it is represented by the following equation;

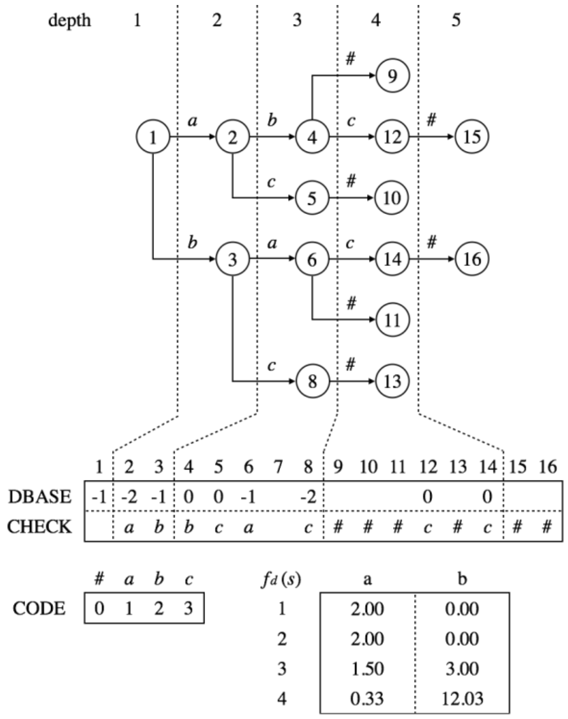



Fig. 3. The double array using linear functions for *K*.

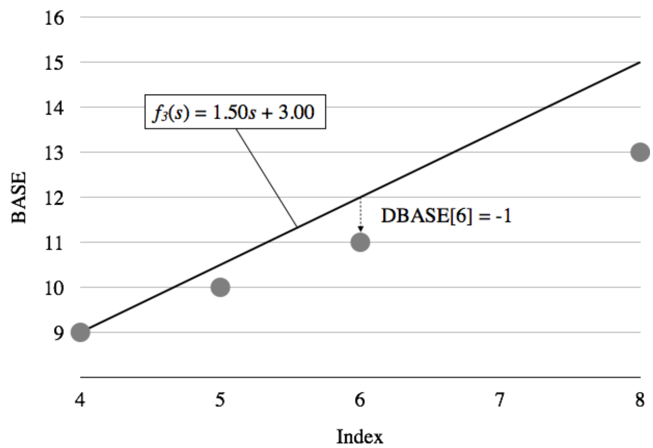



Fig. 4. The scatter diagram of Index-BASE and $f_3(s)$.

$$f_d(s) = a_d s + b_d. \quad (5)$$

In this paper, decimal places of $f_d(s)$ are rounded down. The following equation is established because (4) is the same as (2);

$$\mathrm{DBASE}[s] = \mathrm{BASE}[s] - f_d(s). \quad (6)$$

Moreover, in the same manner of (3), the following equation needs to be satisfied in all pairs of nodes $\{i, j\}$ except the leaf nodes.

$$\mathrm{DBASE}[i] + f_d(i) = \mathrm{DBASE}[j] + f_d(j) \quad (7)$$

DALF is represented by the double array using DBASE and $f_d(s)$. Therefore, its space usage is 3|*D*| bytes. Fig. 3 shows DALF for key set *K*. In Fig. 3, DBASE[*s*] is smaller than BASE values of Fig. 2 because of (6).

Furthermore, Fig. 4 shows a scatter diagram and $f_d(s)$ in depth 3. The scatter diagram has indexes of the double array on the $x$-axis and BASE values on $y$-axis.

## 3.2. Outline of Constructions

DALF defines linear functions for each depth of the trie. At the same time, elements of the double array have blocks in each depth. The blocks are represented by the following equations;

$$smin_d = \begin{cases} 1 & (d = 1) \\ smax_{d-1} + 1 & (d \geq 2) \end{cases}, \tag{8}$$

$$smax_d = \sum_{k=1}^{d} (used_k + unused_k). \tag{9}$$

In (8) and (9), variables in depth $d$ are explained as follows;

1) $smin_d$ is the minimum index.
2) $smax_d$ is the *maximum* index.
3) $used_d$ is the number of valid *elements*.
4) $unused_d$ is the number of invalid *elements*.

For example, because the invalid element in Fig. 3 is 7, $used_3$ and $unused_3$ are 4 and 1, respectively.

Slope $a_d$ and $y$-intercept $b_d$ of $f_d(s)$ are defined by the following equations;

$$\mathrm{a}_d = \frac{used_{d+1}}{used_d + unused_d}, \tag{10}$$

$$\mathrm{b}_d = smax_d + 1 - \mathrm{a}_d smin_d. \tag{11}$$

If DALF cannot be built in (10), slope $a_d$ is increased, BASE[$smin_d$...$smax_d$] is expanded, and DALF is rebuilt in depth $d$. Then, slope $a_d$ is decided again by the following equation;

$$a_d = \frac{used_{d+1}}{used_d + unused_d} + gain \cdot r_d. \tag{12}$$

In (12), $gain \cdot r_d$ is added to slope $a_d$. $gain$ is the addition value for slope $a_d$, and $r_d$ is the number of times to rebuild in depth $d$.

BASE[$s$] is represented by the following equation;

$$\mathrm{BASE}[s] \in \{M_s \cup L_s\}. \tag{13}$$

$M_s$ and $L_s$ are adjusted by using constant $a$. $M_s$ is represented by the following equation;

$$M_s = \{x \mid f_d(s) < x \leq f_d(s) + 2^{16} - \alpha\}. \tag{14}$$

$L_s$ is represented by the following equations;

if $smin_d \leq s < s'$

$$L_s = \{x \mid smax_d + 1 \leq x \leq f_d(s)\}. \tag{15}$$

if $s' \leq s \leq smax_d$

$$L_s = \{x \mid f_d(s) - \alpha \leq x \leq f_d(s)\}. \tag{16}$$

$s'$ is represented by the following equation;

$$s' = \frac{smax_d + 1 - b_d + \alpha}{a_d}. \tag{17}$$

According to (13) - (17), DBASE[$s$] can be represented by 2 bytes.

### 3.3. Construction Algorithms

The construction algorithm of DALF is shown as below;

[**Procedure** *Build*(*BT*)]

**Define:** $S_1 = \ldots = S_{MaxDepth(BT)} := \varnothing$

B-1 *Append*($S_1$, 1)

B-2 *new*[1]:= 1

B-3 **for** *d*:= 1 **to** *MaxDepth(BT)* **do**

B-4 *SetLinear*(*d*)

B-5 *Sort*($S_d$)

B-6 **while** *BuildDepth*(*BT, d*) = False **do**

B-7 *InitDa*(*d*)

B-8 *UpdateLinear*(*d*)

B-9 **end while**

B-10 **end for**

Before procedure *Build*, *BT* is built as a data structure such as a two-dimensional array or a linked list. In *Build*, a variable, an array and functions are used as follows;

Variable $S_d$ stores node numbers of *BT*.

Array *new*[*s*] stores a node number of DALF corresponding to node number *s* of *BT*.

Function *MaxDepth(BT)* returns the deepest depth of *BT*.

Function *Append*($S_d$, *s*) adds *s* to $S_d$.

Function *SetLinear*(*d*) sets $f_d(s)$ using (10) and (11).

Function *Sort*($S_d$) sorts $S_d$ in ascending order of DALF's node number.

Function *InitDa*(*d*) initializes DBASE[$smin_d \ldots smax_d$], CHECK[$smin_{d+1} \ldots smax_{d+1}$], $S_{d+1}$.

Function *UpdateLinear*(*d*) resets $f_d(s)$ using (11) and (12).

In (12), $r_d$ is the number of times to repeat in line B-6 for depth *d*.

In line *B*-1, the root node number is added to *S* in depth 1. The loop of line B-3 builds DALF from *BT* in each depth. The loop of line *B*-6 is repeated until construction of depth *d* is completed. *InitDa* and *UpdateLinear* are called if the construction is failed. In *UpdateLinear,* parameters *gain* and *a* are used.

Function *BuildDepth* in line *B*-6 is shown as below;

[**Function** BuildDepth(BT, d)]

D-1 **for** s **in** $S_d$ **do**

D-2 base := XCheck(s)

D-3 **if** base $\in \{M_{new[s]} \cup L_{new[s]}\}$ **then**

D-4 DBASE[new[s]]:= base – $f_d$(new[s])

D-5 **else**

D-6 **return** False

D-7 **end if**

D-8 **for** *t* **in** Children(*BT*, *s*) **do**

D-9 new[*t*] := base + CODE[Label(*BT*, *t*)]

D-10 CHECK[new[*t*]] := Label(*BT*, *t*)

D-11 **if** Children(*BT*, *t*) $\neq \varnothing$ **then**

D-12 Append($S_{d+1}$, *t*)

D-13 **end if**

D-14 **end for**
D-15 **end for**
D-16 **return** True

Function *BuildDepth* builds DALF in depth $d$ and returns True and False as a result of the construction. In *BuildDepth*, functions are used as follows;

Function *XCheck*($s$) returns the smallest value which satisfies (2), (3) and is over minimum values of $L_{new[s]}$.

Function *Label*($BT, s$) returns a label to destination $s$ in $BT$.

Function *Children*($BT, s$) returns child nodes of $s$ in $BT$.

In line D-3, *base* is checked whether or not to satisfy (13). If *base* does not satisfy (13), *BuildDepth* returns False in order to be rebuilt in depth $d$. The loop of line D-8 traverses all child nodes of $s$ and sets node numbers of DALF and CHECK value. Moreover, the loop prepares to build in depth $d$ + 1.

Table 1. Information about Keyword Sets

| | *K1* | *K2* | *K3* | *K4* |
|---|---|---|---|---|
| Language | English | English | Japanese | Japanese |
| Number of keywords | 1,000,000 | 1,500,000 | 1,000,000 | 1,500,000 |
| Average length (bytes) | 18.5 | 18.5 | 20.9 | 20.8 |
| Minimum length (bytes) | 1 | 1 | 1 | 1 |
| Maximum length (bytes) | 254 | 255 | 241 | 255 |
| File size (MB) | 19.5 | 29.3 | 21.9 | 32.8 |

## 4. Experimental Observations

### 4.1. Problems of Construction Methods for DALF

DALF is built by using parameter *gain* in (12) and parameter $a$ in (14) - (17). The space usage becomes small with reducing *gain*. However, the construction time becomes long, because the number of times to rebuild DALF increases. The space usage becomes large with increasing *gain*. Additionally, the number of times to rebuild DALF is increased because $a_{d+1}$ becomes small in (10). On the other hand, the space usage becomes large with reducing $a$, and the number of times to rebuild DALF increases with increasing $a$. Therefore, these parameters need to be chosen as appropriate values.

However, determinations of these parameters have not defined. In this section, these parameters are observed by experiments, and appropriate parameters are found.

### 4.2. Evaluations

DALF is built by using various combinations of parameters *gain* and $a$. The keyword sets were made by extracting 1,000,000 and 1,500,000 titles from English and Japanese Wikipedia at random. They are called $K_1$ ... $K_4$, and details of $K_1$ ... $K_4$ are shown in Table 1. Japanese keywords including multibyte characters such as Kanji in UTF-8 were used as byte strings. The numerical codes of characters were decided in descending order of appearance frequency in keyword sets. In this experiment, the filling rate of valid elements and the number of times to rebuild are evaluated.

Fig. 5 shows experimental results for $K_1$. When *gain* and $a$ were respectively 0.01 and 24,000, the space usage was the most compact and the filling rate of the valid elements was 97.84%. However, the number of times to rebuild was 16. When *gain* and $a$ are respectively 0.09 and 20,000, the number of times to rebuild was 1. Then, the filling rate of the valid elements was 97.78%.

Fig. 6 shows experimental results for $K_2$. When *gain* and *a* were respectively 0.01 and 8,000, the space usage was the most compact and the filling rate of the valid elements was 94.02%. However, the number of times to rebuild was 63. When *gain* and *a* were respectively 0.11 and 6,000, the number of times to rebuild was 11. Then, the filling rate of the valid elements was 91.59%.

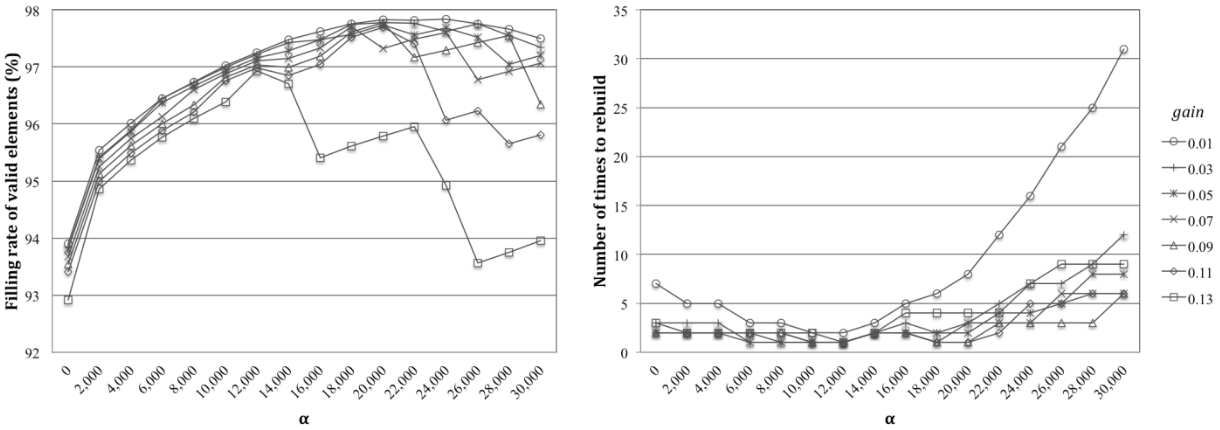



Fig. 5. Experimental results for $K_1$.

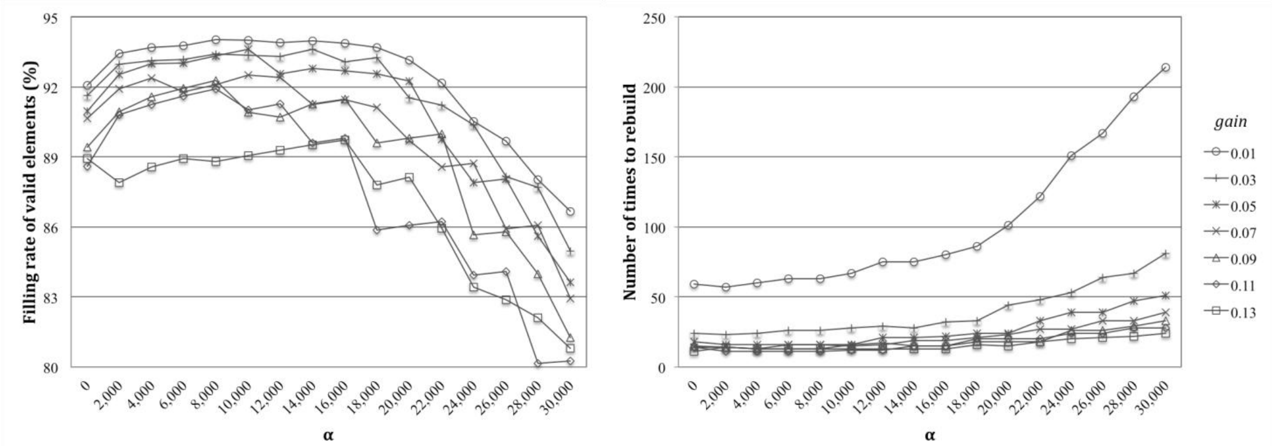



Fig. 6. Experimental results for $K_2$.

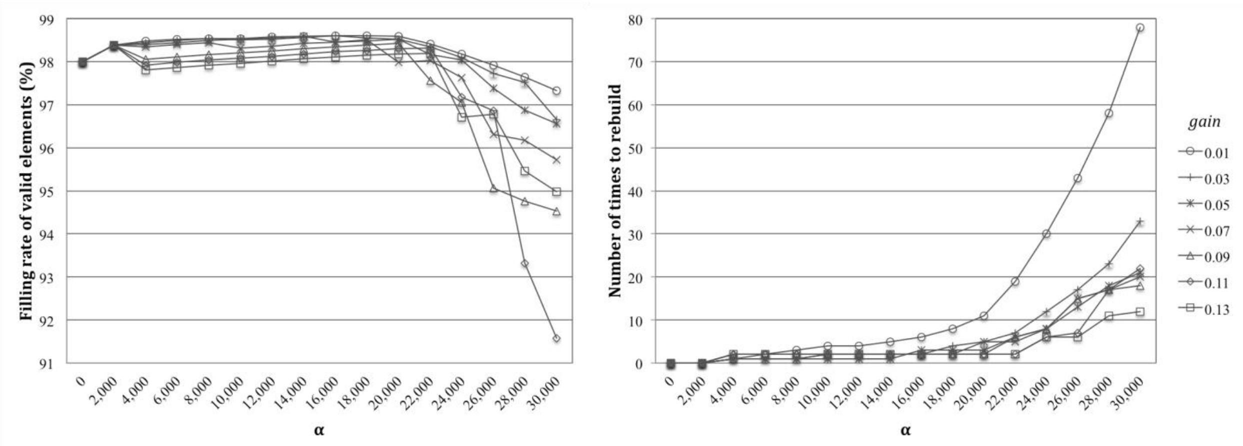



Fig. 7. Experimental results for $K_4$.

In the experimental results for $K_3$, when *a* was from 0 to 24,000, the number of times to rebuild was 0.

When *a* was from 12,000 to 24,000, the filling rate of the valid elements was 98.6%.

Fig. 7 shows experimental results for *K4*. When *gain* and *a* were respectively 0.01 and 16,000, the space usage was the most compact and the filling rate of the valid elements was 98.60%. However, the number of times to rebuild was 6. When *a* was 2,000, the number of times to rebuild was 0. Then, the filling rate of the valid elements was 98.39%.

From the results, it turns out that the space usage becomes the most compact in all keyword sets when *gain* is 0.01. However, the number of times to rebuild increases. In English, the appropriate values of *gain* for the fast construction was 0.09 - 0.11. In Japanese, the appropriate values of α for the fast construction was from 0 to 2,000.

Moreover, it turns out that the appropriate values of α decreases with increasing the number of keywords.

Furthermore, it turns out that Japanese keyword sets can be built DALF more efficiently. In UTF-8, as the first byte represents the length of the following bytes, this first byte frequently appears in keysets of multibyte characters such as Japanese. Therefore, when BASE values are decided, the possibilities of collisions among them become low, the filling rate of the valid elements increases, and the number of times to rebuild DALF decreases. As a result, DALF is more efficient for the keyword sets including multibyte characters such as Japanese, Arabic and Chinese.

## 5. Conclusion

This paper has observed various combinations of parameters *gain* and *a* in DALF. From experiments, it turns out that the space usage becomes compact with decreasing *gain*. In English, the construction speed becomes the fastest when *gain* is 0.09 - 0.11. In Japanese, the construction speed becomes the fastest when *a* is from 0 to 2,000. Furthermore, it turns out that DALF can be built more efficiently by the keyword sets including multibyte characters. A further work is to propose more efficient construction methods of DALF for large keyword sets.

## References


[1] Fredkin, E. (1960). Trie memory. *Communications of the ACM*, *3(9)*, 490–500.

[2] Yang, L., Xu, L., & Shi, Z. (2012). An enhanced dynamic hash TRIE algorithm for lexicon search. *Enterprise Information Systems*, *6(4)*, 419-432.

[3] Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Longman Publishing Co., Inc.

[4] Huang, K., Xie, G., Li, Y., & Liu, A. X. (2011). Offset addressing approach to memory-efficient IP address lookup. *Proceedings of the 2011 IEEE International Conference on Computer Communications* (pp. 306-310).

[5] Navarro, G. (2004). Indexing text using the ziv–lempel trie. *Journal of Discrete Algorithms*, *2(1)*, 87-114, 2004.

[6] Fu, J., Hagsand, O., & Karlsson, G. (2007). Improving and analyzing LC-trie performance for IP-address lookup. *Journal of Networks*, *2(3)*, 18-27.

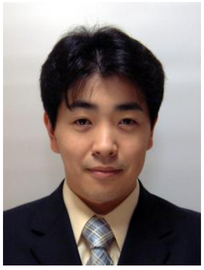
[7] Aoe, J. (1989). An efficient digital search algorithm by using a double-array structure. *Transactions on Software Engineering*, *15(9)*, 1066–1077.

[8] Yata, S., Oono, M., Morita, K., Fuketa, M., & Aoe, J. (2007). A compact static double-array keeping character codes. *Proceedings of the Conference on Information Processing and Management* (pp. 237-247).

[9] Fuketa, M., Kitagawa, H., Ogawa, T., Morita, K., & Aoe, J. (2014). Compression of double array structures for fixed length keywords. *Proceedings of the Conference on Information Processing and Management*

(pp. 796-806).
[10] Kanda, S., Morita, K., Fuketa, M., & Aoe, J. (2014). A compression method of double array structures using approximate straight lines. *IPSJ SIG Technical Report*, 1-6.

**Shunsuke Kanda** received his B.Sc. degree in information science and intelligent systems from University of Tokushima, Japan, in 2014. He is currently a master course student at University of Tokushima. His research interests are information retrieval.



**Kazuhiro Morita** received his B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from University of Tokushima, Japan, in 1995, 1997 and 2000, respectively. Since 2006, he has been a research associate in the Department of Information Science and Intelligent Systems, University of Tokushima, Japan. His research interests are sentence retrieval from huge text databases, double array structures and binary search tree.

**Masao Fuketa** received his B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from University of Tokushima, Japan, in 1993, 1995 and 1998, respectively. He had been a research assistant from 1998 to 2000 in information science and intelligent systems, University of Tokushima, Japan. He is currently an associate professor in the Department of Information Science and Intelligent Systems, University of Tokushima, Japan. He is a member of the information processing society in Japan and the association for natural language processing of Japan. His research interests are information retrieval and natural language processing.

**Jun-Ichi Aoe** received his B.Sc. and M.Sc. degrees in electronic engineering from the University of Tokushima, Japan, in 1974 and 1976, respectively, and received the Ph.D. degree in communication engineering from the University of Osaka, Japan, in 1980. Since 1976, he has been with the University of Tokushima. He is currently a professor in the Department of Information Science and Intelligent Systems, University of Tokushima, Japan. His research interests are natural language processing, a shift-search strategy for interleaved LR parsing, a robust method for understanding NL interface commands in an intelligent command interpreter, and trie compaction algorithms for large key sets. He was the editor of the computer algorithm series of the IEEE Computer Society Press. He is a member of the association for computing machinery and the association for the natural language processing of Japan.