Under consideration for publication in Knowledge and Information Systems

A compression method of double-array structures using linear functions

Shunsuke Kanda, Masao Fuketa, Kazuhiro Morita and Jun-ichi Aoe

Department of Information Science and Intelligent Systems, Tokushima University, Minamijosanjima 2-1, Tokushima 770-8506, Japan

Abstract. A trie is one of the data structures for keyword search algorithms and is utilized in natural language processing, reserved words search for compilers and so on. The double-array and LOUDS are efficient representation methods for the trie. The double-array provides fast traversal at time complexity of $O(1)$, but the space usage of the double-array is larger than that of LOUDS. LOUDS is a succinct data structure with bit-string, and its space usage is extremely compact. However, its traversal speed is not so fast. This paper presents a new compression method of the double-array with keeping the retrieval speed. Our new method compresses the double-array by dividing the double-array into blocks and by using linear functions. Experimental results for varied keywords show that our new method reduced space usage of the double-array up to about 44% and the retrieval speed of the new method was 9-14 times faster than that of LOUDS. Moreover, the results show that the construction speed of the new method was faster than that of the conventional method for a large keyword set.

Keywords: Trie; Double-array; Compression method; Information retrieval

1. Introduction

A trie (Fredkin, 1960) is an ordered tree structure with a character on the edge for keyword retrieval. The term *"trie"* coined by Fredkin was from the word *"retrieval"*. The trie has many functions and is applicative. Therefore, it is used in a broad range of applications to represent a keyword set in fields such as natural language processing (Baeza et al, 1996; Yang et al, 2012), reserved words search for compilers (Aho et al, 2006), text indexing (Navarro, 2004), IP address lookup (Huang et al, 2011) and so on (Aho et al, 1975; Peterson, 1980; Srinivasan

Received Jan 09, 2015
Revised Jul 23, 2015
Accepted Jul 27, 2015

et al, 1998; Brain et al, 1994; Aoe et al, 1996; Fu et al, 2007). Traditionally, the trie was represented either by a two-dimensional array called a matrix form or by a linked-list called a list form. As the matrix form includes many empty elements and is a sparse data structure, its space usage is very large. The list form represents the trie in a more compact form, but its retrieval speed is not so fast.

The double-array presented by Aoe (Aoe, 1989; Aoe et al, 1992; Yata et al, 2007a) is an efficient data structure that represents the trie with two one-dimensional arrays called BASE and CHECK. The double-array provides fast traversal at time complexity of $O(1)$. BASE and CHECK are arrays which represent node numbers. Let n and m be the number of trie nodes and empty elements, respectively. BASE and CHECK elements require $\log(n+m)$ bits[1] each, and the space usage of the double-array is $2(n+m)\log(n+m)$ bits. However, $n+m$ nearly equals to n because m is empirically very small in general. The double-array provides a extremely fast retrieval for the trie, and the memory efficiency is the same as the list form. Therefore, it has been used in many applications (e.g. Darts[2], Darts-clone[3], Chasen[4] and Mecab[5]). To reduce the space usage of the double-array with keeping the retrieval speed, Yata et al. presented Compacted Double-Array (CDA) (Yata et al, 2007b) and Fuketa et al. presented Single-Array with Multi Code (SAMC) (Fuketa et al, 2014). CDA keeps characters in CHECK, and each CHECK element is always represented by $\log\sigma$ bits, where σ denotes the number of character kinds. The space usage of CDA is $(n+m)\log(n+m)+(n+m)\log\sigma$ bits. CDA is more compact than the original double-array because $\log(n+m) > \log\sigma$ in general. SAMC deletes BASE from CDA by using the numerical code matrix corresponding to each character and each depth of the trie. The space usage of SAMC is $(n+m)\log\sigma+h\sigma\log(n+m)$ bits, where h denotes the height of the trie. SAMC becomes compact as the succinct data structures discussed later when h is small, but the method is for fixed length keywords such as a zip code.

To represent tree structures with extremely compact space, many researches have been devoted to so-called succinct data structures for trees. Basically, there are three types of succinct tree representations: Balanced Parentheses sequence (BP) (Jacobson, 1989; Munro et al, 2001), Depth-First Unary Degree Sequence (DFUDS) (Benoit et al, 2005; Jansson et al, 2007) and Level-Order Unary Degree Sequence (LOUDS) (Jacobson, 1989; Delpratt et al, 2006). These represent an ordered tree with $2n+o(n)$ bits. BP and DFUDS are constructed on sequence of balanced parentheses. These provide many operations for the tree such as giving the number of nodes in the subtree of a node and so on. Especially, DFUDS supports more operations in constant time, but BP and DFUDS require complicated additional data structures to operate balanced parentheses, and their space usage is not negligible in practice. To solve this problem, Sadakane et al. proposed a simple representation for balanced parentheses called Fully-Functional succinct tree representation (FF) (Sadakane et al., 2010). When BP and DFUDS are implemented with FF, the space usage of the additional data structures is very

[1] The base of logarithm is 2 throughout this paper.
[2] Darts: Double-ARray Trie System. `http://chasen.org/~taku/software/darts/`
[3] Darts-clone: A clone of the Darts. `https://code.google.com/p/darts-clone/`
[4] ChaSen legacy: an old morphological analyzer. `http://chasen-legacy.sourceforge.jp/`
[5] MeCab: Yet Another Part-of-Speech and Morphological Analyzer. `http://mecab.googlecode.com/svn/trunk/mecab/doc/index.html`

small and the BP and DFUDS become practicable. In (Arroyuelo et al, 2010), Arroyuelo et al. concluded that FF stands out as an excellent practical combination of space occupancy, time performance and functionality. On the other hand, LOUDS is the simplest representation because of not requiring a balanced parentheses representation, but it lacks many basic operations for the tree. However, the trie only needs to traverse to a child node with a character when a keyword is retrieved. In terms of operations for traversing to a child node and getting a label of edge, Arroyuelo et al. concluded that LOUDS excels in (Arroyuelo et al, 2010). LOUDS representation for the trie is more compact than CDA because LOUDS requires $2n+1$ and $n \log \sigma$ bits for the tree structure and labels of edges, respectively. The traversal speed of LOUDS is slower than that of CDA because LOUDS requires to go through extra nodes and to operate bit-string. The experiments in (Fuketa et al, 2014) have shown the results that the retrieval of CDA is much faster than that of LOUDS.

This paper presents a new compression method for the double-array with keeping the retrieval speed. Our new method compresses the space usage of BASE element from $\log(n+m)$ bits to arbitrary x bits by dividing the double-array into blocks and using linear functions. As the new method keeps features of the double-array, its retrieval speed is very fast when compared to LOUDS. In addition, the construction speed of the new method is stable and fast because of dividing the double-array into blocks.

The rest of the paper is organized as follows. Section 2 describes the trie and LOUDS. Section 3 describes the double-array, CDA and the construction algorithms. Section 4 proposes a new compression method of the double-array. Section 5 describes the retrieval and construction algorithms. Section 6 shows theoretical and experimental evaluations. Section 7 concludes this paper and indicates future studies.

2. Trie

2.1. Trie structure

Trie is an ordered tree data structure to store keywords. Fig. 1 shows a trie for keyword set $K = \{$ *"ab"*, *"abc"*, *"ac"*, *"ba"*, *"bac"*, *"bc"*$\}$. In the trie, each keyword is registered as the path from a root node to a leaf node. However, when keywords *"ab"* and *"abc"* are registered in the trie, we can not identify *"ab"* because *"abc"* includes *"ab"* as the prefix. Therefore, special end marker *'#'* is added at the end of keywords to associate each keyword with each leaf node.

A traversal on the trie[6] starts from a root node and retrieves each character in the keyword to a leaf node step by step. Furthermore, the prefixes of keywords are merged. Therefore, the trie can retrieve the longest prefixes at high speed, and the trie provides exact search, common prefix search and predictive search efficiently.

Traditionally, the trie is implemented by the matrix and list form. The matrix form provides fast traversal at time complexity of $O(1)$, but it requires large spaces because it becomes sparse. The list form is more compact, but its retrieval speed is relatively slow because it requires to go through extra nodes. The trie

[6] In this paper, "traversal" in the trie means a transition from a parent node to a child node.

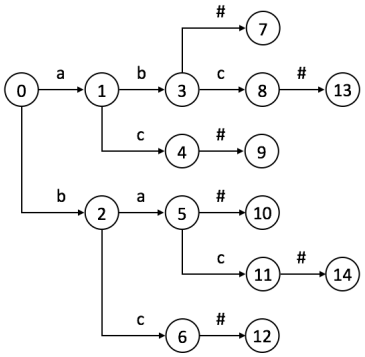

Fig. 1. A trie for keyword set K.

is efficiently represented by the double-array and LOUDS. The double-array can retrieve very fast and is relatively compact. LOUDS is extremely compact, but its retrieval speed is not so fast.

Example 1. In Fig. 1, when *"ba"* is retrieved, the trie traces a route from node 0 (root node) to node 2, node 5, node 10 (leaf node).

2.2. LOUDS structure

As noted in Section 1, we consider LOUDS as the best succinct tree representation for the trie among other succinct data structures. This subsection shows the LOUDS structure for the trie. LOUDS represents a tree structure with a bit-string called LOUDS Bit-String (LBS). Fig. 2 shows LOUDS for keyword set K. LBS is a succinct bit-string represents the number of child nodes according to breadth-first order. Each '1' in LBS corresponds to each node. The first "10" called a super root always stays at the head of LBS as shown in Fig. 2. *rank*/*select* operations are needed to use LBS as a tree structure.

- $rank_b(B, i)$ returns the number of $b \in \{0, 1\}$ up to the i-th position in bit-string B.
- $select_b(B, i)$ returns the position of the i-th $b \in \{0, 1\}$ in bit-string B.

The operations are widely used for bit-string handling.

LOUDS represents the trie by using LBS and an array called LABEL. LABEL stores trie characters. The trie is traversed by following computations, but argument B in *rank*/*select* is omitted for legibility because $B = \text{LBS}$ is obvious.

- LBS position j corresponding to the first child of $\text{LBS}[i] = 1$ is computed with $j = select_0(rank_1(i)) + 1$; However, if $\text{LBS}[j] = 0$, such a node does not exist.
- LBS position j corresponding to the next sibling of $\text{LBS}[i] = 1$ is computed with $j = i + 1$; However, if $\text{LBS}[j] = 0$, such a node does not exist.
- Character c of the edge between the node of $\text{LBS}[i] = 1$ and its parent is computed with $c = \text{LABEL}[rank_1(i) - 1]$.

The space usage of LBS and LABEL are $2n + 1$ and $n \log \sigma$ bits, respectively. LOUDS is more compact than the double-array. In terms of traversal, its time complexity is $O(\sigma)$ when sibling nodes are searched sequentially.

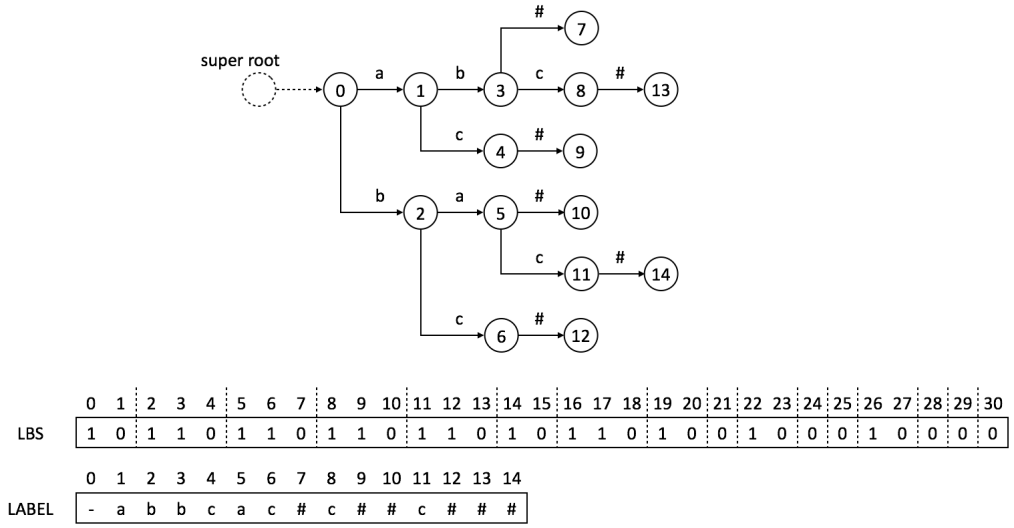

Fig. 2. Trie representation of LOUDS for keyword set K.

Example 2. In Fig. 2, suppose searching for the keyword *"ba"*. First, consider the arc from root node 0 with *'b'*. LBS[0] corresponds to node 0. The LBS position corresponding to the first child of LBS[0] is computed with $select_0(rank_1(0)) + 1 = 2$. Because of LABEL$[rank_1(2) - 1]$ = LABEL[1] = *'a'* $\neq$ *'b'*, the next sibling is checked. The LBS position corresponding to the next sibling of LBS[2] is computed with $2 + 1 = 3$, and the next sibling exists because of LBS[3] = 1. Because of LABEL$[rank_1(3)-1]$ = LABEL[2] = *'b'*, the child node of 0 with *'b'* is computed. In the same manner as the above-mentioned arc, arcs with *'a'* and *'#'* are computed by $select_0(rank_1(3))+1 = 8$, LABEL$[rank_1(8)-1]$ = LABEL[5] = *'a'* and $select_0(rank_1(8)) + 1 = 16$, LABEL$[rank_1(16) - 1]$ = LABEL[10] = *'#'*.

3. Double-array

3.1. Double-array structure

Double-array proposed by Aoe uses two one-dimensional arrays called BASE and CHECK in order to represent trie nodes. Fig. 3 shows a double-array for keyword set K. Basically, each double-array element of BASE and CHECK corresponds to each node in the trie. For example, BASE[s] and CHECK[s] corresponds to node s. The following equations show an arc from node s to node t with character c;

$$\text{BASE}[s] + \text{CODE}[c] = t, \ \text{CHECK}[t] = s. \tag{1}$$

The index of child node t is calculated by the sum of BASE[s] and CODE[c]. CODE[c] is the numerical code of character c, and $0 \leq \text{CODE}[c] < \sigma$. The index of parent node s is stored in CHECK[t].

From Equation (1), the double-array provides the traversal with very few computing, and the traversal time complexity is $O(1)$. The retrieval speed of the double-array is overwhelmingly faster than that of LOUDS. For example, the

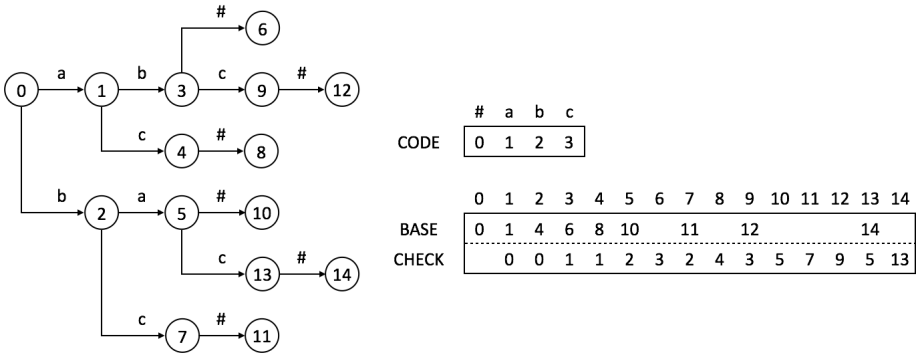

Fig. 3. Trie representation of ODA for keyword set K.

experiments in (Fuketa et al, 2014) have given the result that the double-array can retrieve 12-15 times faster than LOUDS.

The double-array can have empty elements (unused node numbers) because the double-array satisfies Equation (1). Let n and m be the number of trie nodes and empty elements, respectively. BASE and CHECK elements require $\log(n+m)$ bits each because BASE and CHECK values correspond to node numbers. The space usage of the double-array is $2(n+m)\log(n+m)$ bits, but m is a negligible value as compared to n. In this paper, this Original Double-Array method is called ODA.

Example 3. In Fig. 3, suppose searching for the keyword *"ba"*. First, the arc from root node 0 to node 2 with *'b'* is established by BASE[0] + CODE[*'b'*] = 0 + 2 = 2, CHECK[2] = 0. In the same manner as the above-mentioned arc, arcs with *'a'* and *'#'* are established by BASE[2] + CODE[*'a'*] = 4 + 1 = 5, CHECK[5] = 2 and BASE[5] + CODE[*'#'*] = 10 + 0 = 10, CHECK[10] = 5.

3.2. Compacted double-array structure

Compacted Double-Array (CDA) presented by Yata et al. reduces the space usage of ODA. Fig. 4 shows CDA for keyword set K. In this method, characters are stored in CHECK. Therefore, Equation (1) is changed as follows;

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

In ODA, when BASE[s] = BASE[s'] ($s \neq s'$), character c from parent nodes s and s' can traverse to the same child node t. Therefore, CHECK[t] stores the parent node number. However, as CDA stores character c in CHECK[t], the following equation needs to be satisfied in all pairs of internal nodes (s, s');

$$\mathrm{BASE}[s] \neq \mathrm{BASE}[s']. \tag{3}$$

The retrieval speed of CDA is the same as ODA. A character is stored in CHECK, that is to say, CHECK elements is always represented by $\log\sigma$ bits. BASE size is the same as ODA. The space usage of CDA is $(n+m)\log(n+m)+(n+m)\log\sigma$ bits. In general, CDA is more compact than ODA because $\log(n+m) > \log\sigma$.

Example 4. In Fig. 4, suppose searching for the keyword *"ba"*. First, the arc

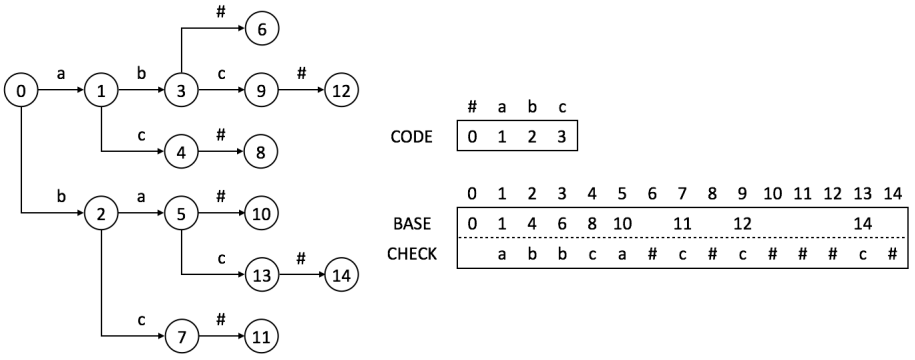

Fig. 4. Trie representation of CDA for keyword set K.

from root node 0 to node 2 with *'b'* is established by BASE[0] + CODE[*'b'*] = 0 + 2 = 2, CHECK[2] = *'b'*. In the same manner as the above-mentioned arc, arcs with *'a'* and *'#'* are established by BASE[2] + CODE[*'a'*] = 4 + 1 = 5, CHECK[5] = *'a'* and BASE[5] + CODE[*'#'*] = 10 + 0 = 10, CHECK[10] = *'#'*.

3.3. Construction algorithm for double-array

When the double-array is constructed, the node numbers are determined with BASE values and CODE values. That is to say, determining the node numbers is the same as determining BASE values. Suppose determining BASE value for node $s \geq 0$. Algorithm 1 shows function XCHECK(A) to determine BASE[s] (A is a character set of arcs from s and A, BASE[s] := XCHECK(A)). Function XCHECK(A) returns minimum *base* such that *is_empty*(*base*+CODE[c]) = TRUE for all characters $c \in A$ (*base*+CODE[c] ≥ 0). CDA satisfies *is_used_base*(*base*) = FALSE. Functions *is_empty* and *is_used_base* are explained as follows;

- *is_empty*(s) returns TRUE if node number s is unused, that is to say, element s of the double-array is empty, otherwise returns FALSE.
- *is_used_base*(*base*) returns TRUE if BASE value *base* is used, otherwise returns FALSE.

When BASE values are determined, the double-array seeks empty elements (unused node numbers). In the original method, empty elements are sought from the head element sequentially (Fig. 5a). When the double-array is constructed with a large keyword set, the construction speed becomes extremely slow. Morita et al. presented methods called skip and link methods in order to speed up the construction (Morita et al, 2001; Morita et al, 2004). The skip method seeks only the rear part of the double-array because the number of empty elements is very small in the front part (Fig. 5b). The skip method determines the start position called *skip_head* for seeking the rear part as the follow;

$$skip\_head := da\_size \cdot skip\_rate, \tag{4}$$

where *da_size* denotes the number of double-array elements and *skip_rate* < 1.0. The skip method requires to determine an appropriate parameter *skip_rate*

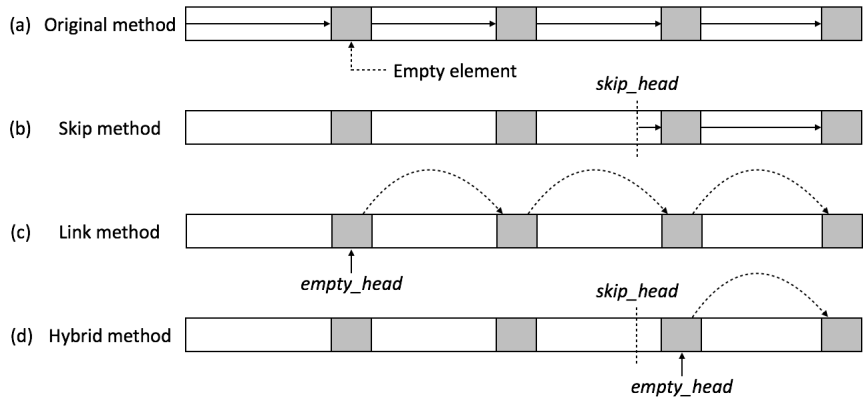

Fig. 5. Examples for seeking empty elements.

because the number of empty elements and the construction speed are in the relation of trade-off.

The link method uses a linked-list called empty-list to link empty elements. The empty-list provides the next empty element, and the link method can seek only empty elements (Fig. 5c). Function XCHECK uses the empty-list. The following variables and function are required to use the empty-list.

- *empty_head* is minimum node number s with $is\_empty(s) =$ TRUE.
- $c_{min} \in A$ is a character such that CODE$[c_{min}]$ is the minimum for all characters in A.
- $next\_empty(s)$ returns a minimum node number u such that $is\_empty(u) =$ TRUE and $u > s$.

Because the link method does not cause the empty elements increase, it is very practicable. Moreover, the construction speed becomes faster by using the hybrid between skip and list methods (Fig. 5d).

Algorithm 1. XCHECK(A): A is a character set.

```
1:  t := empty_head
2:  repeat
3:      base := t − CODE[c_min]
4:      if is_used_base(base) = TRUE then                ▷ For CDA
5:          t := next_empty(t)
6:          continue
7:      end if
8:      flg := TRUE
9:      for all c ∈ A do
10:         if is_empty(base + CODE[c]) = FALSE then
11:             flg := FALSE
12:             break
13:         end if
14:     end for
15:     t := next_empty(t)
16: until flg = TRUE
17: return base
```

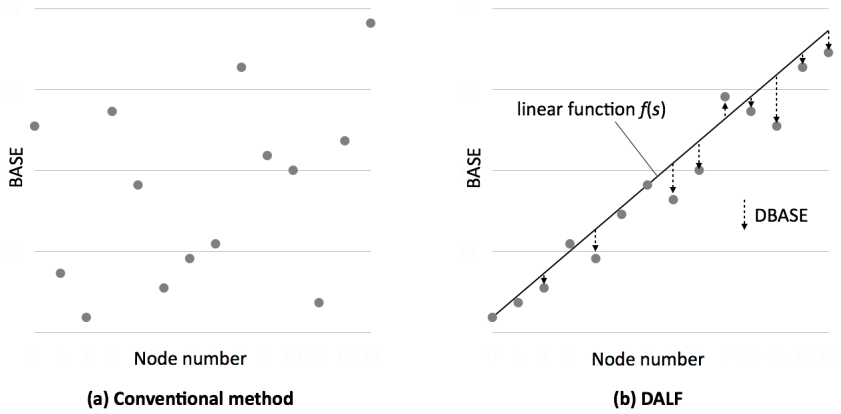

Fig. 6. Examples of scatter diagrams for conventional method and DALF.

The loop from line 2 to line 16 examines appropriate BASE value *base* such that $is\_empty(base + \mathrm{CODE}[c]) = \textsc{true}$ for all characters $c \in A$. Because CDA requires lines 4 and 7 to satisfy Equation (3), there can be more empty elements in CDA than in ODA. The construction speed of CDA is slower than that of ODA.

4. Double-array using linear functions

4.1. Outline

The double-array provides fast traversal at time complexity of $O(1)$. However, BASE array stores values corresponding to destination node numbers, and each BASE element requires $\log(n+m)$ bits. In other words, BASE array causes the double-array size increase. This section proposes Double-Array using Linear Functions (DALF). DALF compresses the BASE element from $\log(n+m)$ bits to $0 < x < \log(n+m)$ bits. Main compression approaches of DALF are as follows;

Step 1: Define linear function $f(s)$ with node number s.
Step 2: Determine BASE[s] near linear function $f(s)$.
Step 3: Use BASE[s] $- f(s)$ as new BASE value for DALF.

In this paper, the new BASE array for DALF is called DBASE. Fig. 6 shows examples of scatter diagrams which have node numbers on the x-axis and BASE values on y-axis. Fig. 6a is the scatter diagram for conventional methods such as ODA and CDA. Fig. 6b is the scatter diagram for DALF. DALF is constructed by determining BASE values near linear function $f(s)$ and using relative values with linear function $f(s)$ instead of BASE values. Therefore, DBASE values become smaller than BASE values. When Step 2 determines BASE[s] such that BASE[s] $- f(s)$ is represented by x bits for all nodes s, DBASE element can be represented by x bits.

The approaches have the problem that BASE[s] $- f(s)$ can not be represented by x bits for a large keyword set. The problem is posed because CDA

satisfies Equations (2) and (3). When BASE[s] − $f(s)$ is not represented by x bits, it is necessary to reconstruct DALF by redetermining linear function $f(s)$ and all BASE values. However, it is difficult to redetermine appropriate linear function $f(s)$, and the construction speed becomes slow. Therefore, this method divides the double-array elements into blocks which are composed of *bsize* elements, and defines linear function $f_b(s)$ for block b (b is a block number which is computed by $\lfloor s/bsize \rfloor$). When BASE[s] − $f_b(s)$ is not represented by x bits, we only need to reconstruct DALF for block b: to redetermine linear function $f_b(s)$ and BASE values in block b. Furthermore, we can easily determine appropriate linear function $f_b(s)$.

As DBASE values and linear function $f_b(s)$ are used instead of BASE values, Equation (2) is changed for DALF as follows;

$$\text{DBASE}[s] + \lfloor f_b(s) \rfloor + \text{CODE}[c] = t,\ \text{CHECK}[t] = c. \tag{5}$$

When $f_b(s)$ is calculated with integers, $f_b(s)$ is calculated as $\lfloor f_b(s) \rfloor$ in following equations. However, the floor functions are omitted for legibility. BASE value is represented by the following equation because Equation (5) is the same as Equation (2).

$$\text{BASE}[s] = \text{DBASE}[s] + f_b(s). \tag{6}$$

In the same manner as Equation (3), the following equation needs to be satisfied in all pairs of internal nodes (s, s');

$$\text{DBASE}[s] + f_b(s) \neq \text{DBASE}[s'] + f_{b'}(s'), \tag{7}$$

where b and b' are computed by $\lfloor s/bsize \rfloor$ and $\lfloor s'/bsize \rfloor$, respectively. $f_b(s)$ is a linear function with node number s, and it is represented by the following equation;

$$f_b(s) = \mathtt{a}_b s + \mathtt{b}_b. \tag{8}$$

DALF provides the double-array by using DBASE and linear function $f_b(s)$. DALF determines BASE[s] such that BASE[s] − $f_b(s)$ is represented by x bits for all nodes s. Therefore, the space usage of DALF is $(n + m)(x + \log \sigma)$ bits. DALF is more compact than CDA because $x < \log(n + m)$.

Example 5. Fig. 7 shows DALF for keyword set K. DBASE values are smaller than BASE values of Fig. 4 because of DBASE[s] = BASE[s] − $f_b(s)$. BASE values are restored by Equation (6). For $s = 9$, BASE[9] = DBASE[9]+$\lfloor f_2(9) \rfloor$ = DBASE[9] + $\lfloor \mathtt{a}_2 \cdot 9 + \mathtt{b}_2 \rfloor = 0 + \lfloor 0.25 \cdot 9 + 13.00 \rfloor = 15$.

4.2. Definition for construction method

We discuss definitions for DALF by using T_b which denotes an interval of node numbers traversed from block b. T_b has fields min and max, where $T_b.min$ and $T_b.max$ denote the minimum and maximum node numbers traversed from block b, respectively. In other words, T_b is represented as follows;

$$T_b = [T_b.min, T_b.max].$$

Because T_b is an interval, it can include unused node numbers. T_b is not defined when block b does not include internal nodes, and the T_b is used as $\emptyset$. Moreover,

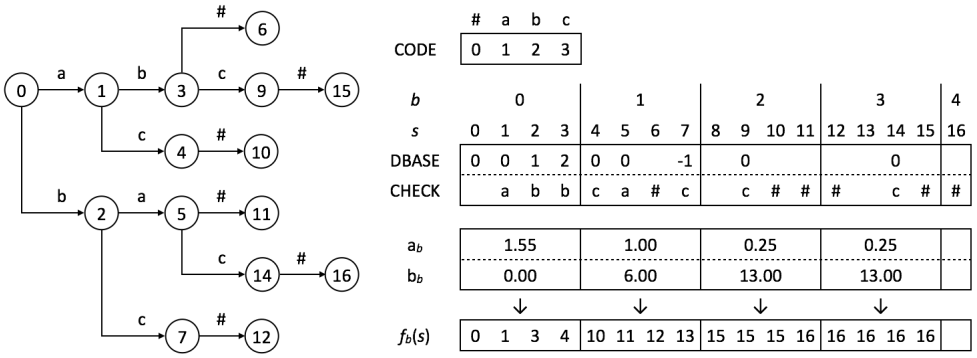

Fig. 7. Trie representation of DALF for keyword set K ($bsize = 4$).

the following set is defined for discussions;

$$T_{0\cdots b} := \bigcup_{k=0}^{b} T_k.$$

The following equation holds because the first block $b = 0$ always includes a root node.

$$T_{0\cdots b} \neq \emptyset.$$

To denote the minimum and maximum indices in block b, s_min_b and s_max_b are used. That is to say, $[s\_min_b, s\_max_b]$ denotes the interval of indices in block b, and the following equations hold;

$$s\_min_b = b \cdot bsize,\ \ s\_max_b = (b+1) \cdot bsize - 1.$$

In this paper, a construction of the double-array for each block indicates a determination of BASE values for each block in order of increasing block number step by step. After $T_{0\cdots b-1}$ is determined, BASE values in block $b > 0$ are determined. To construct DALF for each block efficiently, BASE values are determined by obeying the follows;

Definition 1. $T_{0\cdots b-1}.max < T_b.min$ $(0 < b)$.

Definition 2. For any node with a non-zero block number, its child nodes have strictly larger block numbers.

In this regard, we exclude the case that block b does not include an internal node. When Definition 1 is obeyed, the following equation holds;

$$T_b \cap T_{b+1} = \emptyset. \tag{9}$$

Because of Definition 1, we can easily reconstruct DALF because we only need to initialize BASE$[s_min_b, s_max_b]$ and CHECK$[T_b.min, T_b.max]$ when we begin to reconstruct DALF in block b. Moreover, we can simply define linear function $f_b(s)$ without depending on nodes in other blocks. In Section 4.3, Equation (9) is well used when linear function $f_b(s)$ is defined. On the other hand, Definition 2 is obeyed in $s\_min_{b+1} \leq T_b.min$ $(b > 0)$ to determine BASE values of each block step by step. However, Definition 2 omits the first block $b = 0$ because the first block consists of only a root node when the first block obeys Definition 2.

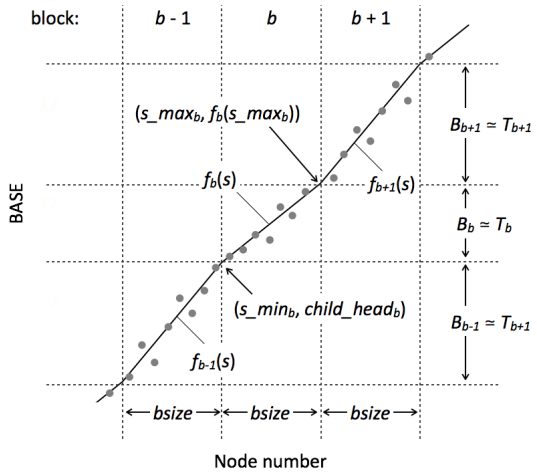

Fig. 8. Examples of scatter diagram when Definitions 1 and 2 are obeyed.

Therefore, DALF requires distinct construction methods for the first block and the others. Fig. 8 shows the scatter diagram such as Fig. 6 when Definitions 1 and 2 are obeyed. The details of Fig. 8 are shown in Section 4.3.

To discuss definitions for DALF, we define the following equation;

$$child\_head_b := \begin{cases} 0 & (b = 0) \\ max(T_{0\cdots b-1}.max + 1, s\_min_{b+1}) & (b > 0), \end{cases} \tag{10}$$

where max returns the maximum value in the arguments. When node number t traversed from block b is determined such that $child\_head_b \leq t$, Definitions 1 and 2 are obeyed. In this paper, $child\_head_b$ is very important because it is used in the definition of linear function and so on.

4.3. Definition of linear function

DALF divides the double-array elements into blocks which are composed of *bsize* elements, and constructs the double-array for each block like Fig. 8. BASE values are determined near linear function $f_b(s)$ to represent DBASE values by x bits. DALF defines linear function $f_b(s)$ as the line through point $(s\_min_b, child\_head_b)$ and an increasing linear function. The definitions of slope $\mathtt{a}_b$ and y-intercept $\mathtt{b}_b$ are shown in this subsection. However, suppose that block b includes internal nodes.

When linear function $f_b(s)$ is determined, it is important to determine BASE values in block b near linear function $f_b(s)$. Let B_b be the interval of BASE values in block b. When BASE values in block b are determined near linear function $f_b(s)$,

$$[child\_head_b, s\_max_b] \simeq B_b.$$

Because we can consider $\mathrm{BASE}[s] + \mathrm{CODE}[c] = t$ as $\mathrm{BASE}[s] \simeq t$ in $n \gg \sigma$, the

equation holds;

$$B_b \simeq T_b.$$

That is to say, the equation holds;

$$[child\_head_b, f_b(s\_max_b)] \simeq T_b. \tag{11}$$

If slope $\mathtt{a}_b$ is too large, $f_b(s\_max_b)$ becomes too big and $|T_b|$ increases. As a result, T_b includes many unused node numbers. If slope $\mathtt{a}_b$ is too small, $|T_b|$ decreases and we can not determine BASE values in block b near linear function $f_b(s)$. When T_b does not include unused node numbers and we can determine BASE values in block b near linear function $f_b(s)$, the slope $\mathtt{a}_b$ is the best parameter. Suppose that T_b does not include unused node numbers, the equation holds;

$$|T_b| = total\_degree_b, \tag{12}$$

where $total\_degree_b$ denotes the total degree of nodes in block b. Because Equation (9) holds by obeying Definition 1, Equation (12) holds. From Equation (11), linear function $f_b(s)$ becomes appropriate in

$$|[child\_head_b, f_b(s\_max_b)]| = total\_degree_b.$$

Therefore, slope $\mathtt{a}_b$ is defined as following equation;

$$\mathtt{a}_b := \frac{|[child\_head_b, f_b(s\_max_b)]|}{bsize} = \frac{total\_degree_b}{bsize} \quad (b > 0). \tag{13}$$

Because of Definition 2, $total\_degree_b$ is not changed after linear function $f_b(s)$ is determined. On the other hand, $total\_degree_0$ can not be calculated because Definition 2 omits the first block $b = 0$. Thus, we define $\mathtt{a}_0$ by using the average degree in the trie. Let ave_degree be the average degree of internal nodes in the trie, we consider $total\_degree_0$ as $ave\_degree \cdot bsize$. Slope $\mathtt{a}_0$ is defined by the follow equation;

$$\mathtt{a}_0 := \frac{ave\_degree \cdot bsize}{bsize} = ave\_degree. \tag{14}$$

Example 6. In Fig. 7, $total\_degree_1 = 4$ because degrees of node 4, 5, 6 and 7 are 1, 2, 0 and 1, respectively. Therefore, $\mathtt{a}_1 = total\_degree_1/bsize = 4/4 = 1.00$.

Y-intercept $\mathtt{b}_b$ is defined by using slope $\mathtt{a}_b$ and the point $(s\_min_b, child\_head_b)$. From Equation (8), y-intercept $\mathtt{b}_b$ is defined as the follow equation;

$$\mathtt{b}_b := child\_head_b - \mathtt{a}_b \cdot s\_min_b. \tag{15}$$

Example 7. In $b = 1$ of Fig. 7, $child\_head_1 = max(T_0.max + 1, s\_min_2) = max(9 + 1, 8) = 10$ because the traversal from node 3 to node 9 is defined. Therefore, $\mathtt{b}_1 = child\_head_1 - \mathtt{a}_1 \cdot s\_min_1 = 10 - 1.00 \cdot 4 = 6.00$.

4.3.1. Reconstruction

If DALF can not determine BASE[s] that BASE[s] $- f_b(s)$ is represented by x bits, it is necessary to redetermine BASE values in block b after slope $\mathtt{a}_b$ increases and T_b expands. Slope $\mathtt{a}_b$ is redetermined by the following equation;

$$\mathtt{a}_b := \mathtt{a}'_b + gain \cdot r_b, \tag{16}$$

where *gain* is the addition value for slope $\mathtt{a}_b$, r_b is the number of reconstruction in block b and $\mathtt{a}'_b$ is the initial parameter determined by Equation (13) or Equation (14). If parameter *gain* is too large, T_b expands too much and the number of empty elements in block b increases. If parameter *gain* is too small, T_b expands little and r_b increases. As a result, the construction speed becomes slow. Therefore, parameter *gain* needs to be defined as an appropriate value.

Furthermore, we give the proof that the reconstruction using Equation (16) can always terminate for all tries. Here, the upper bound of r_b is shown by discussing the upper bound of slope $\mathtt{a}_b$. To discuss the upper bound of slope $\mathtt{a}_b$, suppose the following cases;

Case 1. In terms of the number of trie nodes and the size of DBASE element, the worst cases for block b are shown the follows;

–Block b does not include an unused node number, and the nodes are all internal nodes.

–The degree of each node is σ, where σ is the maximum degree.

–$x = 1$ bit, that is to say BASE$[s] = f_b(s)$.

As Case 1 is the worst case in DALF, the slope $\mathtt{a}_b$ for Case 1 is the upper bound. In Case 1, each node has σ child nodes, and the number of internal nodes in block b is *bsize*. When linear function $f_b(s)$ is ignored, the following equation holds about T_b;

$$|T_b| = bsize \cdot \sigma. \tag{17}$$

Because node numbers can be determined freely without depending on nodes in other blocks from Equation (9), the child node numbers traversed from block b are determined in consecutive. Hence, Equation (17) holds. As $bsize \cdot \sigma$ is the maximum value of $|T_b|$ for all tries, the slope $\mathtt{a}_b$ is the upper bound when linear function $f_b(s)$ satisfies Equation (17) in BASE$[s] = f_b(s)$. When BASE values are determined in order of increasing node number, the BASE values satisfying Equation (17) are shown as follows;

$$\text{BASE}[s\_min_b + i] = child\_head_b + i \cdot \sigma \ (0 \leq i < bsize). \tag{18}$$

The linear function $f_b(s)$ satisfies Equation (18) in BASE$[s] = f_b(s)$ when slope $\mathtt{a}_b$ and y-intercept $\mathtt{b}_b$ are defined as follows;

$$\mathtt{a}_b = \sigma, \ \mathtt{b}_b = child\_head_b - s\_min_b \cdot \sigma. \tag{19}$$

From Equation (19), the upper bound of slope $\mathtt{a}_b$ is σ, and the upper bound of r_b is shown as the follow;

$$r_b = \left\lceil \frac{\sigma - \mathtt{a}'_b}{gain} \right\rceil \tag{20}$$

The discussion shows that the reconstruction can always terminate in the worst case such as Case 1. In Equation (20), values σ, $\mathtt{a}'_b$ and *gain* do not depend on the number of trie nodes. Therefore, the reconstruction using Equation (16) can always terminate for an exceedingly large trie.

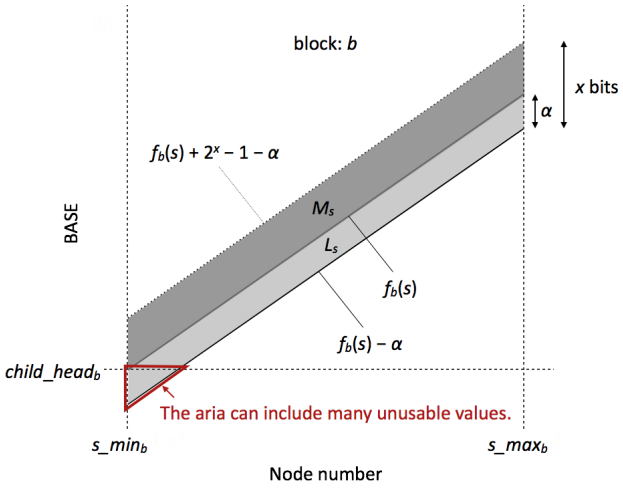

Fig. 9. An example for ranges of BASE values in block b.

4.4. Interval of BASE values

DALF represents $\mathrm{BASE}[s] - f_b(s)$ by x bits for all nodes s in order to represent $\mathrm{DBASE}[s]$ by x bits. This subsection shows intervals of BASE values when DBASE values are represented by x bits. The interval of $\mathrm{BASE}[s]$ is defined by the following equation;

$$\mathrm{BASE}[s] \in (M_s \cup L_s). \tag{21}$$

Fig. 9 shows an example for intervals of BASE values in block b.

M_s and L_s are intervals of BASE values; higher and lower BASE values than linear function $f_b(s)$ are M_s and L_s, respectively. However L_s includes $f_b(s)$. M_s and L_s are adjusted by using parameter α $(0 \leq \alpha < 2^x - 1)$. M_s is represented as the following equation;

$$M_s = \{y \mid f_b(s) < y < f_b(s) + 2^x - 1 - \alpha\}. \tag{22}$$

Because DALF requires to represent empty elements of DBASE such as DBASE[6] in Fig. 7, M_s does not include $f_b(s) + 2^x - 1 - \alpha$. In other words, the maximum value of DALF is used to represent empty elements. L_s is represented as the following equation;

$$L_s = \{y \mid f_b(s) - \alpha \leq y \leq f_b(s)\}. \tag{23}$$

In terms of values in L_s smaller than $child\_head_b$, many unusable BASE values can be included because of the following reasons;

- To obey Definitions 1 and 2, DALF determines node number t traversed from block b such that $child\_head_b \leq t$. Because $\mathrm{BASE}[s] + \mathrm{CODE}[c] = t$ from Equation (2), $child\_head_b \leq \mathrm{BASE}[s] + \mathrm{CODE}[c]$. From the equation, BASE values smaller than $child\_head_b - \mathrm{CODE}[c]$ can not be used in block b.
- When BASE values smaller than $child\_head_b$ are used in block $b' < b$, the BASE values can not be used in block b.

Therefore, L_s can include many unusable values when α is big and $|L_s|$ is large.

On the other hand, when $|L_s|$ decreases with decreasing α, BASE values become big and $|T_b|$ becomes large, that is to say, the number of empty elements increases. Therefore, parameter α requires to be defined as appropriate value.

5. Algorithms for DALF

5.1. Retrieval algorithm

Function RETRIEVE(str) in Algorithm 2 searches keyword str on DALF. The function returns TRUE if str is found, otherwise returns FALSE. When a new method is compared with conventional methods, one difference in their retrieval algorithms is calculation of a destination node number. $str[k]$ denotes the k-th character for str, and $len(str)$ denotes the length of keyword str.

Algorithm 2. RETRIEVE(str): str is a keyword such that $str[len(str)] =$ *'#'*.

```
1: s := 0                                              ▷ a root node
2: for k := 1 to len(str) do
3:     t := DBASE[s] + f_b(s) + CODE[str[k]]           ▷ b = ⌊s/bsize⌋
4:     if CHECK[t] ≠ str[k] then
5:         return FALSE
6:     end if
7:     s := t
8: end for
9: return TRUE
```

The loop from line 2 to line 8 traverses arcs with k-th character of keyword str. In line 3, destination node t from node s with character $str[k]$ is calculated by Equation (5). If the arc does not exist, the function returns FALSE in line 5. If the function can traverse nodes with str, the function returns TRUE in line 9.

Example 8. Suppose searching for the keyword *"ba"* in Fig. 7. This case is the same as the call of function RETRIEVE with the argument $str =$ *"ba#"*. First, the arc from root node 0 to node 2 with $str[1] =$ *'b'* is calculated by DBASE[0] + $f_0(0)$ + CODE[*'b'*] = 0 + 0 + 2 = 2, CHECK[2] = *'b'* in lines 3 and 4. In the same manner as the above-mentioned arc, arcs with $str[2] =$ *'a'* and $str[2] =$ *'#'* are established by DBASE[2] + $f_0(2)$ + CODE[*'a'*] = 1 + 3 + 1 = 5, CHECK[5] = *'a'* and DBASE[5] + $f_1(5)$ + CODE[*'#'*] = 0 + 11 + 0 = 11, CHECK[11] = *'#'*. In line 9, the function returns TRUE.

5.2. Construction algorithm

Construction algorithms for DALF are presented in this subsection. In the algorithms, after all keywords are registered to the trie called a base trie, node numbers of a new trie for DALF are determined. The base trie is implemented by a link form or a matrix form. The algorithms construct DALF for each block step by step, and determine node number s for the new trie that DBASE[s] is represented by x bits for all nodes s. Definitions 1 and 2 are obeyed.

The base trie is denoted by BT and has the following functions;

– $BT.root()$ returns a root node number in base trie BT.

– $BT.labels(v)$ returns an arc's characters set from node v in base trie BT.
– $BT.child(v, c)$ returns a child node number traversed from parent node v with c in base trie BT.

BT is traversed by using min-priority queue q with 2-tuple (v, s), where v and s are node numbers of BT and the new trie, respectively. Min-priority queue q gives priority to node number s of the new trie. That is to say, the algorithms traverse BT in order of increasing s. q has the following functions;

– $q.empty()$ returns TRUE if q is empty, otherwise returns FALSE.
– $q.enque((v, s))$ enqueues (v, s).
– $q.deque()$ dequeues (v, s).

In the algorithms, BT and q are used globally.

In the construction algorithms, function YCHECK$_b$ in Algorithm 3 is invoked when BASE value in block b is determined. The functions and c_{min} in Section 3.3 are used. The following $empty\_head_b$ is used instead of $empty\_head$;

– $empty\_head_b$ is minimum node number s such that $is\_empty(s)$ = TRUE and $child\_head_b \leq s$.

Function YCHECK$_b(A, lower)$ returns minimum BASE value $base$ in block b such that $base \geq lower$ and $is\_empty(base + \mathrm{CODE}[c])$ = TRUE $(c \in A)$. Moreover $is\_used\_base(base)$ = FALSE.

Algorithm 3. YCHECK$_b(A, lower)$: A is a character set, $lower$ is a lower bound of the BASE value.

```
 1: t := empty_head_b
 2: repeat
 3:     base := t − CODE[c_min]
 4:     if base < lower or is_used_base(base) = TRUE then
 5:         t := next_empty(t)
 6:         continue
 7:     end if
 8:     flg := TRUE
 9:     for all c ∈ A do
10:         if is_empty(base + CODE[c]) = FALSE then
11:             flg := FALSE
12:             break
13:         end if
14:     end for
15:     t := next_empty(t)
16: until flg = TRUE
17: return base
```

With the same technique as function XCHECK, function YCHECK$_b$ uses the empty-list. In DALF, because node numbers smaller than $child\_head_b$ are not used to obey Definitions 1 and 2, YCHECK$_b$ only needs to seek subsequent empty element of $child\_head_b$ by using $empty\_head_b$ (Fig. 10). This method is equal to the hybrid method in Fig. 5d. The following construction algorithms is complex when compared to ODA and CDA, but the construction speed is fast because YCHECK$_b$ provides the hybrid method naturally.

The construction algorithms consist of two main parts for the first block and

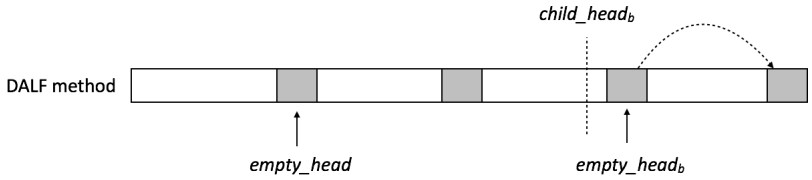

Fig. 10. An example for seeking of empty elements for DALF

for the other blocks. To construct the new trie for the first block 0, function ConstructFirstBlock in Algorithm 4 is invoked. Function ConstructFirstBlock begins to construct with a root node, and the new trie is constructed in order of increasing node number for the new trie by using q. When the function finishes to determine BASE values in the first block, the function returns TRUE. If the BASE value does not satisfy Equation (21), the function returns FALSE. In the algorithm, $L_s.min$ denotes the minimum value in L_s.

Algorithm 4. ConstructFirstBlock()

```
1: q.enque((BT.root(), 0))
2: while q.empty() = FALSE and ⌊s/bsize⌋ = 0 for q[0] = (v, s) do
3:     (v, s) := q.deque()
4:     base := YCheck_0(BT.labels(v), L_s.min)
5:     if base ∉ (M_s ∪ L_s) then
6:         return FALSE
7:     end if
8:     DBASE[s] := base − f_0(s)
9:     for all c ∈ T.labels(v) do
10:        t := base + CODE[c]
11:        CHECK[t] := c
12:        if c ≠ '#' then
13:            q.enque((T.child(v, c), t))
14:        end if
15:    end for
16: end while
17: return TRUE
```

In line 1, root node numbers are enqueued to begin the construction. The loop from line 2 to line 16 traverses BT and constructs the new trie in the first block. Line 4 determines BASE value by using YCheck. If the BASE value does not satisfy Equation (21) in line 5, line 6 returns FALSE. That is to say, DBASE value determined in line 8 is represented by x bits. The loop from line 9 to line 15 calculates the new node number traversed from s and determines new CHECK values. Moreover, child node numbers are enqueued to traverse BT.

On the other hand, function ConstructBlock in Algorithm 5 is invoked to construct the new trie for block $b > 0$. Function ConstructBlock(P) receives set P which stores node numbers (v, s) in block b, and constructs the new trie in block b.

Algorithm 5. ConstructBlock(P): P is a set which stores node numbers in block b.

```
1: Q := ∅
2: for all (v, s) ∈ P do
```

```
3:     base := YCheck_b(T.labels(v), L_s.min)                ▷ b = ⌊s/bsize⌋
4:     if base ∉ (M_s ∪ L_s) then
5:         return FALSE
6:     end if
7:     DBASE[s] := base − f_b(s)
8:     for all c ∈ T.labels(v) do
9:         t := base + CODE[c]
10:        CHECK[t] := c
11:        if c ≠ '#' then
12:            Append (T.child(v, c), t) to Q
13:        end if
14:    end for
15: end for
16: q.enque((v, s)) for all elements (v, s) ∈ Q
17: return TRUE
```

In function ConstructBlock, set Q with (v, s) are used to store node numbers traversed from block b temporarily. The loop from line 2 to line 15 constructs the new trie in block b by traversing nodes in P. In the same manner as function ConstructFirstBlock, DBASE and CHECK values are determined. On the other hand, node numbers traversed from block b are appended to Q in line 12. Because of line 12, we do not require to remove the node numbers from q when BASE value does not satisfy Equation (21) in line 4. When all BASE values in block b satisfy Equation (21), line 16 enqueues the node numbers by using Q, and line 17 returns TRUE.

Procedure Construct in Algorithm 6 constructs the new trie for each block by using functions ConstructFirstBlock and ConstructBlock. In procedure Construct, $total\_degree_b$ and ave_degree are given from BT. Variable da_size denotes the number of elements of DALF like Equation (4). All elements and q are initialized in advance.

Algorithm 6. Construct()

```
1: child_head_0 := 0
2: Determine a_0 and b_0 using Equations (14) and (15), respectively
3: r_0 := 0
4: while ConstructFirstBlock() = FALSE do
5:     Initialize all DBASE and CHECK elements and q
6:     r_0 := r_0 + 1
7:     Update a_0 using Equation (16)
8: end while
9: while q.empty() = FALSE do
10:    P := ∅
11:    b := ⌊s/bsize⌋ for q[0] = (v, s)
12:    while q.empty() = FALSE and ⌊s/bsize⌋ = b for q[0] = (v, s) do
13:        (v, s) := q.deque() and append (v, s) to P
14:    end while
15:    child_head_b := max(da_size + 1, s_min_{b+1})
16:    Determine a_b and b_b using Equations (13) and (15), respectively
17:    r_b := 0
18:    while ConstructBlock(P) = FALSE do
19:        Initialize DBASE[s_min_b, s_max_b] and CHECK[child_head_b, da_size]
```

Table 1. Theoretical observations.

Method	Time complexity for retrieval	Space usage [bits]
ODA	k	$64(n+m)+8\sigma$
CDA	k	$40(n+m)+8\sigma$
DALF	k	$(x+8)(n+m)+64(n+m)/bsize+8\sigma$
LOUDS	$k\sigma$	$10.75n+1.375$

20: $r_b := r_b + 1$
21: Update $\mathtt{a}_b$ using Equation (16)
22: **end while**
23: **end while**

In lines 1 to 8, the new trie in the first block 0 is constructed. The loop from line 4 to line 8 repeats until BASE values in block 0 satisfies Equation (21). When the construction fails, slope $\mathtt{a}_0$ is updated using Equation (16). The loop always terminates from Equation (20). The loop from line 9 to line 23 constructs the new trie for each block $b > 0$ by using function ConstructBlock. The loop from line 12 to line 14 moves node numbers in block b from q to P. In line 15, da_size is the same as $T_{0\cdots b-1}.max$. The loop from line 18 to line 22 constructs the new trie for block b by using P. In the same manner as the first block, slope $\mathtt{a}_b$ is updated and the new trie is reconstructed if BASE values in block b does not satisfy Equation (21). In line 19, $[T_b.min, T_b.max] \subseteq [child\_head_b, da\_size]$ because $child\_head_b \leq T_b.min$ and $da\_size = T_b.max$. Moreover, $T_{0\cdots b}.max < child\_head_b$ because of Definition 1. Therefore, CHECK$[child_head_b, da_size]$ includes CHECK values traversed from only block b, and we can easily initialize CHECK values because of constructing DALF for each block.

6. Evaluations

6.1. Theoretical observations

Table 1 shows the theoretical observations for ODA, CDA, DALF and LOUDS. The retrieval algorithm among ODA, CDA and DALF are very similar. The time complexities of retrieval for them are $O(k)$, where k denotes the length of the input keyword. As all characters can be checked, the time complexity of LOUDS is $O(k\sigma)$, where σ denotes the number of character kinds, that is to say, the maximum degree in the trie. Therefore, LOUDS is slower than other methods.

In terms of the space usage, we observe the allocations in Section 6.2. n and m denote the number of trie nodes and empty elements, respectively. Table 2 shows the information about the keyword sets for experiments in Section 6.2. In Table 2, the maximum number of trie nodes was greater than 2^{24} and less than 2^{32} even in an extremely large keyword set such as English Wikipedia. Therefore, an integer with 32 bits is sufficient to represent node numbers, and suppose that each integer in the methods is represented by 32 bits. Because multibyte characters such as Kanji were used as byte strings, $\sigma \leq 2^8$, and suppose that each character can be represented by 8 bits.

In ODA, BASE and CHECK elements are represented by 32 bits each. In CDA, BASE and CHECK elements are represented by 32 and 8 bits, respectively. In DALF, DBASE and CHECK elements are represented by x ($0 < x < 32$) and 8 bits, respectively. Because we allocated 32 bits each to $\mathtt{a}_b$ and $\mathtt{b}_b$, each

Table 2. Information about the keyword sets and the tries.

	WordNet	URL	Japanese Wiki	English Wiki
Information about the keyword set				
File size [MB]	4.02	12.10	32.30	227.19
Number of keywords	147,306	219,516	1,518,205	11,519,354
Total length	1,692,291	12,306,787	32,357,551	226,714,224
Maximum length	71	880	253	255
Minimum length	1	14	1	1
Average length	11.48	56.06	21.31	19.68
Information about the tries				
Number of trie nodes	879,563	5,936,065	16,590,101	110,962,030

linear function $f_b(s)$ requires 64 bits. As the number of blocks is calculated by $(n+m)/bsize$[7] , the total space usage of linear functions $f_b(s)$ is $64(n+m)/bsize$ bits. When x is small and $bsize$ is big, the space usages of DBASE and $f_b(s)$ decrease, but m can increase. x, $bsize$ and m are evaluated by Section 6.2. The number of CODE elements is σ and CODE array requires 8σ bits. In terms of LOUDS, the space usage is calculated from Tx-library[8] which is a popular open-source straightforward implementation of the trie using LOUDS. As noted in Section 2.2, LBS and LABEL requires $2n+1$ and $8n$ bits, respectively. In addition, the extra space for $rank/select$ operations requires $0.375(2n+1)$ bits. The total space usage of LOUDS is $10.75n + 1.375$ bits. The observation results show that LOUDS is more compact than ODA and CDA. The space usage of DALF depends on x, $bsize$ and m.

6.2. Experimental observations

ODA, CDA and DALF were implemented with C++, and Tx-library was used for LOUDS. Experiments for space usage, retrieval speed and construction speed were executed on the following PC; Quad-Core Intel Xeon 2 x 2.4 GHz (L2 cache: 256 KB). The following keyword sets were used;

– English words of WordNet-2.3.0[9]
– URL set
– Page titles of Japanese Wikipedia[10]
– Page titles of English Wikipedia[11]

Table 2 shows information about the keyword sets. Multibyte characters such as Kanji in UTF-8 were used as byte strings. The numerical codes of characters were determined in order of descending appearance frequency in the keyword sets (Liu et al, 2011). The allocations are the same as Table 1. All methods are constructed via a link-form trie.

[7] Strictly speaking, the number of blocks is $\lceil (n+m)/bsize \rceil$, but the ceil function is omitted for simplicity. Likewise, the extra space for $rank/select$ operations is calculated in LOUDS.
[8] Tx: Succinct Trie Data structure. `https://code.google.com/p/tx-trie/`
[9] WordNet 3.0. `http://wordnetcode.princeton.edu/3.0/WordNet-3.0.tar.gz`
[10] jawiki dump progress on 20150118. `http://dumps.wikimedia.org/jawiki/20150118/jawiki-20150118-all-titles-in-ns0.gz`
[11] enwiki dump progress on 20150205. `http://dumps.wikimedia.org/enwiki/20150205/enwiki-20150205-all-titles-in-ns0.gz`

First, Fig. 11 shows experimental results for parameters x, *bsize*, α and *gain* in terms of the space usage, construction time and the number of reconstruction. We conducted the experiments by changing parameters and constructing DALF. We set $x = 8, 16$. To compute $b = \lfloor s/bsize \rfloor$ at high speed by using bit-shift, we set *bsize* as a power of 2 ranging in $[32, 4096]$. We set $\alpha = 64, 128, 192$ in $x = 8$ and $\alpha = 16384, 32768, 49152$ in $x = 16$. We set $gain = 1.0, 1.5$. The number of reconstruction in $x = 16$ was almost 0, and there were rarely different among the results. Therefore, Fig. 11 shows only the result for $\alpha = 32768$ and $gain = 1.0$ in $x = 16$.

Totally, the performances in $x = 8$ rapidly decrease from $bsize = 512, 1024$ because the number of reconstruction rapidly increases, but the performances in $x = 16$ do not decrease much because the number of reconstruction is almost 0. From Equation (20), the number of reconstruction for each block decreases with increasing *gain*. However, the number of reconstruction for total blocks can increase with increasing *gain* because the number of elements also increases. In terms of the space usage, $x = 8$ is about 70% of $x = 16$ in $bsize = 256, 512$. $\alpha = 128, 192$ becomes relatively small when $bsize \leq 128$ and $x = 8$, but $\alpha = 64$ becomes relatively small when $1024 \leq bsize$ and $x = 8$. In $1024 \leq bsize$, $gain = 1.0$ provides small space usages. The construction times in $bsize = 256, 512$ are largely unchanged among the parameters. In conclusion, DALF provides a good performance when $x = 8$ and $bsize = 256, 512$. In terms of α and *gain* in the case, the performances do not change in $64 \leq \alpha \leq 192$ and $1.0 \leq gain \leq 1.5$.

Second, Fig. 12 shows experimental results of comparison among ODA, CDA, DALF and LOUDS in terms of the space usage, retrieval time and construction time. DALF were examined in $x = 8, 16$ bits. We set $bsize = 512$, $\alpha = 128, 32768$ and $gain = 1.0$. ODA was constructed by the list method in Fig. 5c, but CDA was constructed by the hybrid method in Fig. 5d because the construction speed becomes extremely slow in the list method. We set $skip\_rate = 0.98$ for the hybrid method in Equation (4). We conducted the experiments for the raw keyword sets and the 10-90% subsets. 100% on x-axis denotes the raw keyword sets.

In terms of the space usage, we can see that DALF is valid to reduce space usage for the double-array. In $x = 16$, the space usage is about 63-65% when compared to CDA. In $x = 8$, the space usage is about 44-45% when compared to CDA. The retrieval speed of DALF is about 2.6-3.5 times slower than ODA and CDA because the traversal of DALF requires more computations. However, DALF can retrieve about 9-14 times faster than LOUDS because DALF keeps the features of the double-array. When the link method is used, the construction speed for the double-array methods depends largely on the number of times to seek empty elements. ODA is faster than other double-array methods because empty elements are easily used and *empty_head* is frequently updated. On the other hand, *empty_head* of CDA is infrequently updated and its construction speed is slow because CDA requires to satisfy Equation (3). When the hybrid method in Fig. 5d is used, the construction speed becomes fast depending on *skip_rate*, but the space usage increases with increasing *skip_rate*. The construction algorithm of DALF is the most complex, but its construction speed becomes fast like CDA because YCHECK_b naturally provides the hybrid method.

In detail, we discuss the comparison between CDA and DALF in terms of the construction speed. Because CDA and DALF are constructed by using the hybrid method, the construction speeds of CDA and DALF depend on the number of subsequent elements of *skip_head* or $child\_head_b$. The construction speed becomes slower when increasing the number of subsequent elements, but the

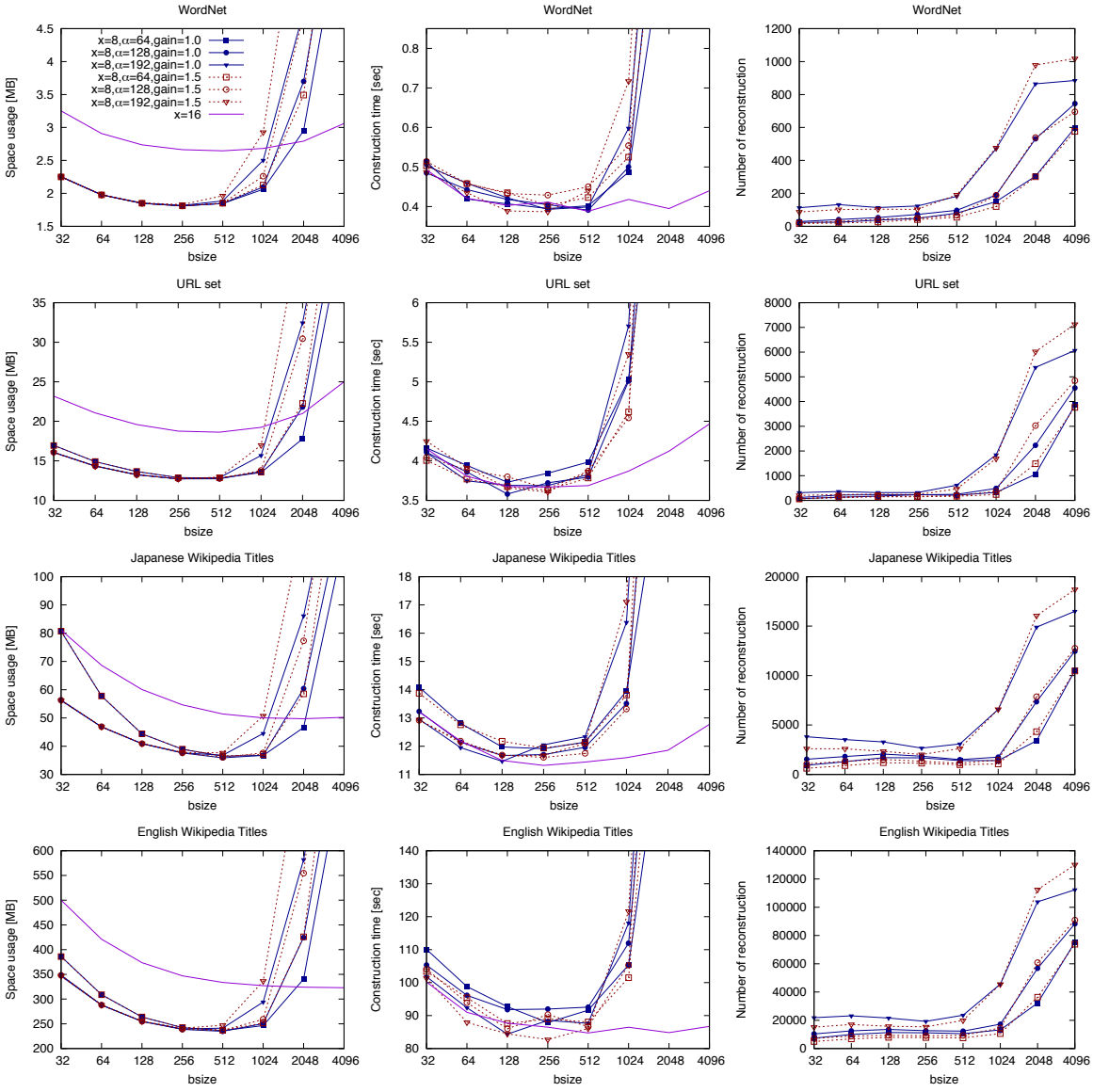

Fig. 11. Experimental results for parameters of DALF. In terms of parts that we can not see result of the graph, the space usages in $x = 8, \alpha = 128, 192$ are relatively small except WordNet, when $bsize = 32, 64, 128$.

space usage becomes larger when decreasing the number of subsequent elements. In CDA, the number of subsequent elements is $da\_size \cdot (1.0 - skip\_rate)$ from Equation (4). As da_size depends on the size of the trie, the number of subsequent elements in CDA increases with increasing the size of the trie. On the other hand, the subsequent elements in DALF depend on T_b. As the maximum $|T_b|$ is $bsize \cdot \sigma$ from Equation (17), the maximum number of the subsequent elements

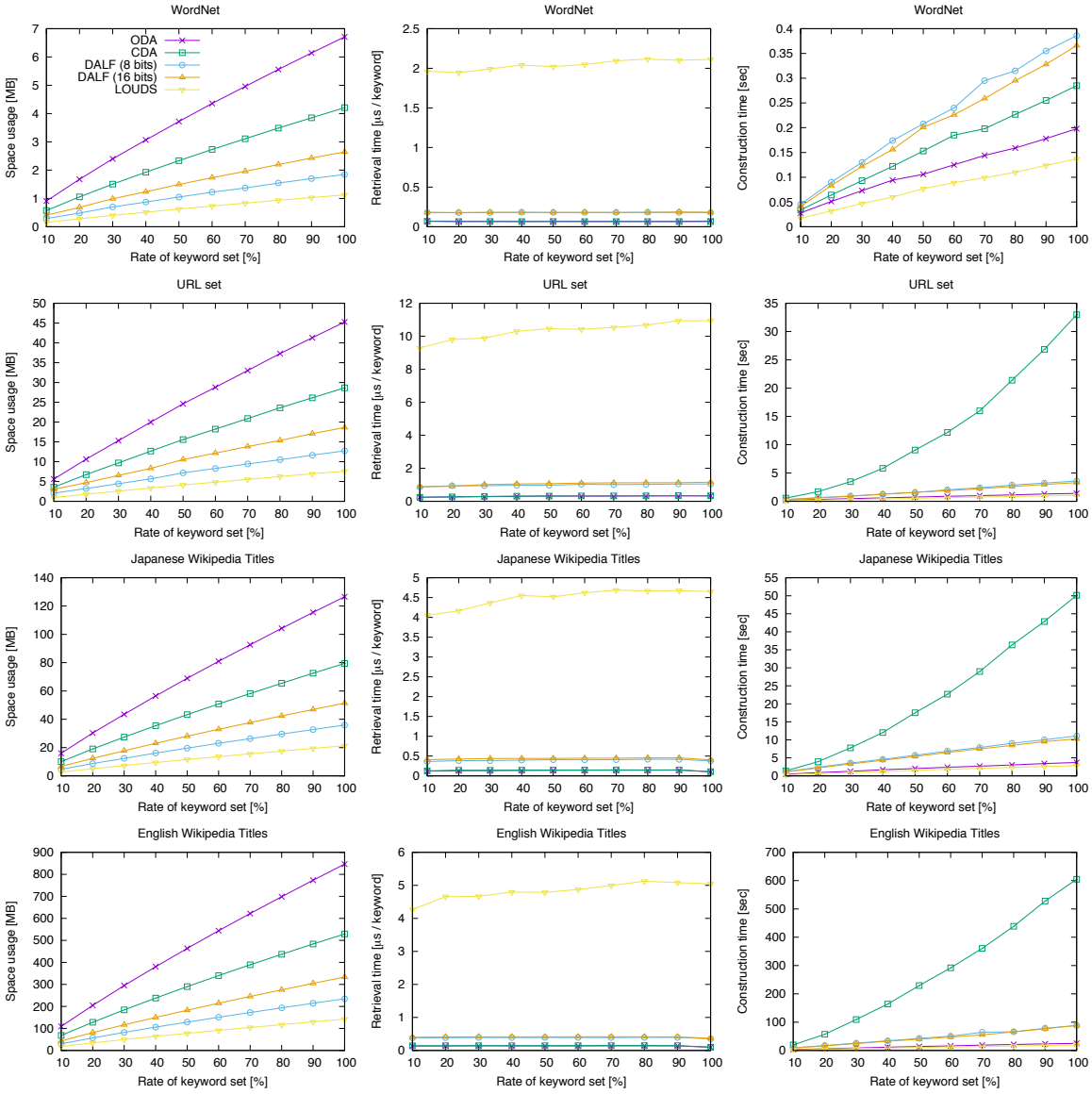

Fig. 12. Experimental results of comparison among ODA, CDA, DALF and LOUDS. In retrieval time, ODA and CDA are overlapping, and DALF methods are also overlapping. In construction time except WordNet, ODA and LOUDS are overlapping, and DALF methods are also overlapping.

in DALF is $bsize \cdot \sigma$ which is independent from the size of the trie. In WordNet, the construction speed of CDA is faster than that of DALF because the size of the trie is small and the number of the subsequent elements in CDA is small. However, in other keyword sets, the construction speed of DALF is faster than that of CDA because the size of the trie is big. Needless to say, the construction speed of CDA becomes faster by increasing *skip_rate* or fixing the number of the subsequent elements, but the space usage also becomes big. DALF always provides a compact trie and a stable and fast construction in $bsize = 256, 512$. In the experiments, the construction speed of DALF is about ten times faster than that of CDA. From the above mentioned, DALF is the most reasonable among the methods.

7. Conclusion

This paper has presented DALF, a compression method of the double-array by dividing the elements into blocks and defining linear functions. From theoretical and experimental observations, it is verified that DALF is more effective than other double-array methods. DALF reduces the space usage of CDA by up to about 44%. Moreover, the retrieval speed of DALF is 9-14 times faster than that of LOUDS. The construction speed of DALF is faster than that of CDA for a large keyword set. However, we could not compress DALF smaller than LOUDS in the experiments. The future study will propose a double-array method smaller than LOUDS.

References

Aho A V, Corasick M J (1975) Efficient string matching: An aid to bibliographic search. *Commun. ACM* 18(6): 333–340

Aho A V, Lam M S, Sethi R, et al (2006) Compilers: Principles, Techniques, and Tools (2Nd Edition). Addison-Wesley Longman Publishing, Boston, MA, USA, Chapters 3 and 4

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

Aoe J, Morimoto K, Sato T (1992) An efficient implementation of trie structures. *Software: Practice and Experience* 22(9): 695–721

Aoe J, Morimoto K, Shishibori M, et al (1996) A trie compaction algorithm for a large set of keys. *IEEE Transactions on Knowledge and Data Engineering* 8(3): 476–491

Arroyuelo D, Cnovas R, Navarro G, et al (2010) Succinct Trees in Practice. In: *ALENEX*, pp. 84–97

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

Benoit D, Demaine E D, Munro J I, et al (2005) Representing Trees of Higher Degree. *Algorithmica* 43: 275–292

Brain M, Tharp A (1994) Using tries to eliminate pattern collisions in perfect hashing. *IEEE Transactions on Knowledge and Data Engineering* 6(2): 239–247

Delpratt O, Rahman N, Raman R (2006) Engineering the louds succinct tree representation. In: *Proceedings of WEA 2006*, pp. 134–145

Fredkin E (1960) Trie memory. *Commun. ACM* 3(9): 490–499

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

Fuketa M, Kitagawa H, Ogawa T, et al (2014) Compression of double array structures for fixed length keywords. *Information Processing & Management* 50(5): 796–806

Huang K, Xie G, Li Y, et al (2011) Offset addressing approach to memory-efficient IP address lookup. In: *Proceedings IEEE INFOCOM*, pp. 306–310

Jacobson G (1989) Space-efficient static trees and graphs. In: *30th Annual Symposium on Foundations of Computer Science*, pp. 549–554
Jansson J, Sadakane K, Sung W (2007) Ultra-succinct representation of ordered trees. In: *ACM-SIAM Symposium on Discrete Algorithms*, pp. 575–584
Liu H, Nuo M, Ma L, et al (2011) Compression methods by code mapping and code dividing for chinese dictionary stored in a double-array trie. In: *IJCNLP*, pp. 1189–1197
Morita K, Fuketa M, Yamakawa Y, et al (2001) Fast insertion methods of a double-array structure. *Software: Practice and Experience* 31(1): 43–65
Morita K, Atlam E, Fuketa M, et al (2004) Fast and compact updating algorithms of a double-array structure. *Information Sciences* 159(12): 53–67
Munro J, Raman V (2001) Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing* 31: 762–776
Navarro G (2004) Indexing text using the zivlempel trie. *Journal of Discrete Algorithms* 2(1): 87–114
Peterson J (1980) Computer programs for spelling correction: An experiment in program design. Springer, Berlin Heidelberg, pp. 1–129
Sadakane K, Navarro G (2010) Fully-functional succinct trees. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 134–149
Srinivasan V, Varghese G, Suri S, et al (1998) Fast and scalable layer four switching. In: *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pp. 191–202
Yang L, Xu L, Shi Z (2012) An enhanced dynamic hash trie algorithm for lexicon search. *Enterprise Information Systems* 6(4): 419–432
Yata S, Oono M, Morita K, et al (2007a) An efficient deletion method for a minimal prefix double array. *Software: Practice and Experience* 37(5): 523–534
Yata S, Oono M, Morita K, et al (2007b) A compact static double-array keeping character codes. *Information Processing & Management* 43(1): 237–247

Author Biographies

Shunsuke Kanda received B.Sc. degree in information science and intelligent systems from Tokushima University, Japan, in 2014. He is currently a master course student at Tokushima University. He is a student member of the information processing society in Japan. His research interests are string processing, database engineering and information retrieval.

Masao Fuketa received B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1993, 1995 and 1998, respectively. He had been a research assistant from 1998 to 2000 in information science and intelligent systems, Tokushima University, Japan. He is currently an associate professor in the department of information science and intelligent systems, Tokushima University, 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.

Kazuhiro Morita received B.Sc., M.Sc. and Ph.D. degrees in information science and intelligent systems from Tokushima University, Japan, in 1995, 1997 and 2000, respectively. He had been a research assistant from 2006 to 2014 in information science and intelligent systems, Tokushima University, Japan. He is currently an associate professor in the department of information science and intelligent systems, Tokushima University, Japan. His research interests are sentence retrieval from huge text databases, double array structures and binary search tree.

Jun-ichi Aoe received B.Sc. and M.Sc. degrees in electronic engineering from Tokushima University, Japan, in 1974 and 1976, respectively, and received the Ph.D. degree in communication engineering from Osaka University, Japan, in 1980. Since 1976, he has been with Tokushima University. He is currently a professor in the department of information science and intelligent systems, Tokushima University, 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.

Correspondence and offprint requests to: Shunsuke Kanda, Department of Information Science and Intelligent Systems, Tokushima University, Minamijosanjima 2-1, Tokushima 770-8506, Japan. Email: shnsk.knd@gmail.com