Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 4, April 2019, pp. 1247-1254
DOI: 10.23940/ijpe.19.04.p20.12471254

Compression Method of Factor Oracle by Triple-Array Structures

Koji Bando[a], Takato Nakano[b], Kazuhiro Morita[c], and Masao Fuketa[c,*]

[a]*NTT Plala Inc., 24th Florr, Sunshine 60, 3-1-1 Higashi Ikebukuro, Toshima-ku, Tokyo, 170-6024, Japan*
[b]*Nichia Corporation, 491 Oka, Kaminaka-Cho, Anan-Shi, Tokushima, 774-8601, Japan*
[c]*Tokushima University, 2-1 Minami-Josanjima, Tokushima, 770-8506, Japan*

Abstract

Pattern matching is an important technique in text processing and is used for character string replacement and search. A factor oracle is a data structure for pattern matching, and it is a finite state automaton that can search substrings. This data structure consists of internal and external transitions and has the characteristic property of accepting at least all substrings. The automaton including the factor oracle is represented by a two-dimensional array called Table, Johnson method, etc. Search using Table is fast, but the memory capacity is large. The Johnson method has the feature of representing the factor oracle using a small amount of storage, and since it can be achieved with a transition speed of O(1), it is considered as fast as Table. The memory of the Johnson method depends on the size of the element itself and the number of elements. In other words, by reducing these, the applicability of the factor oracle to enormous data will be further enhanced. In this research, using compact double-array, we propose a method consisting of compressing the size of the elements themselves, improving the Johnson method in order to achieve a construction using external transitions only, and compressing the number of elements. As shown by the experimental results, the proposed method has higher storage efficiency compared to the Johnson method and is capable under certain conditions of high-speed search.

Keywords: factor oracle; pattern matching; triple-array; double-array

(Submitted on March 12, 2019; Revised on March 14, 2019; Accepted on April 15, 2019)

1. Introduction

In recent years, with the popularization of computing machines and networks that were subject to price reduction and capacity enlargement, the amount of accumulated text data has increased, and therefore fast and advanced search is required. There are various searching techniques [1-3], one of which is pattern matching [4]. Pattern matching is a technique of searching for patterns from text data, and it is used for character string replacement and search. The data structures used for pattern matching include the suffix tree [5] and the factor oracle [6]. The suffix tree has the disadvantage of taking up enormous amounts of computation space in exchange for high-speed search. For that reason, it is not suitable for large-scale pattern matching.

On the other hand, the factor oracle is a very simple and space efficient acyclic automaton generated from arbitrary finite character strings and can be constructed in linear time. This data structure consists of internal and external transitions, and it has the characteristic property of accepting at least all substrings. Studies on musical improvisations that make use of this property have been conducted [7-8]. Other research includes using the factor oracle to extract repeated suffixes and repetitive substrings, which are applied to gene sequence analysis and data compression. Automatons like the factor oracle can be represented in several ways, such as two-dimensional array structures (Table), a triple-array structure by Johnson [9], etc. Search using Table is fast because it can be achieved with a transition speed of O(1). However, since a one-dimensional array must be allocated to each node of the automaton, the increasing memory capacity becomes a disadvantage. On the other hand, the Johnson method has the feature of representing the factor oracle using a small amount of storage, and since it can be achieved with a transition speed of O(1), it is considered as fast as Table. The memory of the Johnson method depends on the size of the element itself and the number of elements. In other words, by reducing these, the applicability of the factor oracle to enormous data will be further enhanced. Hence, in this research, using compact double-array [10], we

* Corresponding author.
E-mail address: fuketa@is.tokushima-u.ac.jp

propose a method consisting of compressing the size of the elements themselves, improving the Johnson method in order to achieve a construction using external transitions only, and compressing the number of elements.

The rest of the paper is organized as follows. Section 2 describes the factor oracle and the data structure. Section 3 explains how the triple-array is improved using the proposed method, the procedures of the compression, and the search algorithm. Section 4 evaluates the validity of the proposed method through verification experiments and includes a discussion. Section 5 summarizes the results of this research.

2. Factor Oracle

2.1. Overview of the Factor Oracle

Pattern matching is the problem of finding the first or all the positions where a pattern appears in different string. Although pattern matching can be considered a simple problem, it can be applied in various situations, and it is very important to require a data structure that achieves an efficient application.

For a given string *p*, the factor oracle is a deterministic finite automaton that accepts at least all the factors (substrings) of *p*. The factor oracle consists of internal transitions that transits characters in the given order of strings and other external transitions. Figure 1 shows the factor oracle of the string *p* = "abbbaab". In Figure 1, the straight arrows represent the internal transitions, while the curved arrows represent the external transitions.

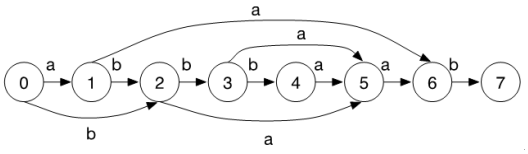

Figure 1. Factor oracle for "abbbaab"

The factor oracle of a string *p* of length *m* has the following properties:

(1) Acyclic.
(2) All nodes are final nodes.
(3) Accepts at least all factors of *p*.
(4) Number of nodes is (*m* + 1).
(5) Number of transitions is greater than *m* and less (2*m* – 1).

According to properties (1)-(3), the factor oracle has no self-loop, and since all nodes are in a final state, the construction algorithm becomes very simple. It also has the characteristic of accepting words that are not factors of *p*. For example, in the factor oracle of Figure 1, besides all the factors of *p*, words that are not factors of *p* such as "aba" and "abba" are accepted. Furthermore, from the properties (4) and (5), the number of nodes of the factor oracle is always (*m* + 1), which makes the number of nodes and the number of transitions extremely small.

2.2. Representation Method of the Factor Oracle

Since the factor oracle is a deterministic finite automaton, we explain two data structures used to represent automata.

2.2.1. Table

For automata, the simplest method with a transition search computational complexity of O(1) is Table, which represents transitions using two-dimensional arrays. For Table, in order to preserve the transition destinations for all characters in each node, assuming the number of nodes is *n* and the character type is σ, a total of σ × *n* double-arrays is required. In general, however, the number of transitions coming out of a node is much less than σ, and most elements of the two-dimensional array are not used. Table becomes an inappropriate method for representing automata for large-scale texts because the larger the text is, the more memory is wasted.

Figure 2 shows the Table representation to Figure 1. In Figure 2, the column indicates the node number and the row indicates the transition character. For example, in Figure 1, since a transition is made from node 3 to node 5 by the character 'a', 5 is stored to table['a'][11].

	0	1	2	3	4	5	6	7
a	1	6	5	5	5	6		
b	2	2	3	4			7	

Figure 2. Table for Figure 1

2.2.2. Johnson Method

The Johnson method is a method that consists of compressing and storing the state transition table of an automaton by using three one-dimensional arrays called BASE, CHECK, and NEXT. It is used in language processing systems such as YACC [12]. For CODE being the function that returns the internal representation value of a character, the transition by the transition character *label* from node number *s* to node number *t* is defined as follows:

$$\begin{aligned} pos &= \mathrm{BASE}[s] + \mathrm{CODE}(label) \\ s &= \mathrm{CHECK}[pos] \\ t &= \mathrm{NEXT}[pos] \end{aligned} \tag{1}$$

This equation uses the sum of BASE[*s*] + CODE(*label*) to verify whether CHECK[*pos*] is equal to the transition source node number *s*. The destination node *t* is determined by NEXT[*pos*]. Figure 3 shows the Johnson method, provided that the internal representation values for 'a' and 'b' are 1 and 2, respectively. For example, in Figure 3, when transitioning from node number 3 by 'a', we have BASE[3] + 'a' = 5 + 1 = 6. We confirm that the value obtained matches the value of the transition source node number using CHECK[6] = 3, and we determine the destination node using NEXT[6] = 5.

	0	1	2	3	4	5	6	7	8	9	10	11
BASE	-1	1	3	5	7	8	9					
CHECK	0	0	1	1	2	2	3	3	4	5		6
NEXT	1	2	6	2	5	3	5	4	5	6		7

Figure 3. Johnson method for Figure 1

In the Johnson method, the number of BASE's elements is the number of nodes. Since CHECK and NEXT match the number of edges, the space efficiency is extremely high compared to Table. In addition, since each character transition is achieved using O(1), this makes it an excellent method capable of performing high-speed searching.

2.3. Construction

The online construction algorithm of the factor oracle using Table is shown in reference [6]. When reading a given string *p* from left to right, this algorithm constructs while updating the automaton for every character it reads, which makes it able to construct extremely fast. The BASE's index in the Johnson method corresponds to the node number, and the value stored in NEXT also represents the destination node number. Therefore, we can easily switch from the construction algorithm of Table to the construction algorithm of the Johnson method's automaton.

A method consisting of deleting NEXT from the Johnson method and creating DAWG (Directed Acyclic Word Graph) using the two arrays BASE and CHECK has been proposed [13]. A method consisting of deleting BASE from the Johnson method and creating an automaton using the two arrays CHECK and NEXT has also been proposed [14]. However, in these methods, the index of BASE or CHECK represents the edge number and not the node number [13], which makes the construction complicated. It is also very difficult to apply it to the online construction algorithm of the factor oracle, and its execution speed becomes very slow.

3. Compression Method of the Triple-Array for Factor Oracle

3.1. Compression of the Array CHECK

Among the methods of expressing automata where the child node does not have multiples parent nodes (such as the trie) is the double-array [11]. The double-array uses the two one-dimensional arrays BASE and CHECK to express the transitions in a trie. The array CHECK has the node number of the transition source stored in it. There is a compact double-array [10]

proposed by Yada et al. that aims to compress the array CHECK and thus express a more compact double-array. The compact double-array compresses each element size to 1 byte by storing transition characters in CHECK. This method maintains a search speed equivalent to the double-array and expresses the trie with higher storage efficiency.

	0	1	2	3	4	5	6	7	8	9	10	11
BASE	-1	1	3	5	7	8	9					
CHECK	a	b	a	b	a	b	a	b	a	a		b
NEXT	1	2	6	2	5	3	5	4	5	6		7

Figure 4. Compact triple-array for Figure 3

It is also possible to incorporate the CHECK array compression of this compact double-array into Johnson method. Figure 4 shows the array resulting from the compression of the array CHECK of the Johnson method in Figure 3. By incorporating the compact double-array, it is possible to reduce the size by storing characters instead of numbers in each element of CHECK. Using this array, the transition by the transition character *label* from node number *s* to node number *t* is defined as follows:

$$\begin{aligned} pos &= \mathrm{BASE}[s] + \mathrm{CODE}(label) \\ label &= \mathrm{CHECK}[pos] \\ t &= \mathrm{NEXT}[pos] \end{aligned} \tag{2}$$

Equation (2) uses *pos* as the sum of BASE[*s*] + CODE(*label*) to verify whether *label* = CHECK[*pos*] and determines the destination using *t* = NEXT[*pos*]. For example, in Figure 4, when transitioning from node number 3 by 'a', we have BASE[3] + 'a' = 5 + 1 = 6. Then, we check if the value matches the transition character using CHECK[6] = 'a', and we determine the destination node using NEXT[6] = 5.

3.2. Reducing the Number of Elements of the Array CHECK

In the Johnson method and the method mentioned in Section 3.1, after verifying the transition using CHECK, the next destination is determined using NEXT. This is because, in an automaton, there are cases where a transition is made to the same node by multiple characters. However, in a factor oracle, there is always one type of transition character to a certain destination. For example, in Figure 1, all transition characters to node 5 are by 'a'. Therefore, by storing the one type transition character to a certain destination node, after determining the next destination node in advance, it is possible to confirm the transition character to that destination node. In other words, Equation (2) is modified as follows:

$$\begin{aligned} pos &= \mathrm{BASE}[s] + \mathrm{CODE}(label) \\ t &= \mathrm{NEXT}[pos] \\ label &= \mathrm{CHECK}[t] \end{aligned} \tag{3}$$

In this equation, NEXT first determines the next destination, and CHECK confirms the transition character to that destination. In order to make it applicable to this equation, the modified array of Figure 4 is shown in Figure 5. For example, since the transition character to node 5 is 'a', CHECK[5] = 'a' is stored. Using this feature, when transitioning from node number 3 by 'a', we have BASE[3] + 'a' = 6 + 1 = 7. The next destination node is determined by NEXT[7] = 5, and lastly, by verifying that the value matches the transition character using CHECK[5] = 'a', we can judge that the transition by 'a' from node 3 to node 5 is correct. Because of this data structure, the number of CHECK's elements matches the length of the string *p*, which makes it smaller than in the method of Section 3.1.

	0	1	2	3	4	5	6	7	8	9	10	11	12
BASE	0	2	4	6	8	9	10						
CHECK		a	b	b	b	a	a	b					
NEXT		1	2	6	2	5	3	5	4	5	6		7

Figure 5. Triple-array by Equation (3)

3.3. Reduction of the Number of Elements of the Array NEXT

In the data structure of Section 3.2, the transition character from node 0 to node 1 is stored in CHECK[1], and the transition character from node 1 to node 2 is stored in CHECK[2]. In the factor oracle of string *p*, when searching for *p* (length *m*) from node 0, the transition is made in the order of node 0 -> 1 -> 2-> ⋯ -> *m*. In other words, since CHECK[1] to CHECK[*m*] stores the string *p* as it is, it is possible to verify the internal transition by comparing the CHECK value of the node number following to the current node number and the actual transition character. Therefore, the array NEXT of internal transitions becomes unnecessary, and it is enough to construct the array of external transitions only. The array obtained from external transitions only is shown in Figure 6. BASE becomes only the transition source of the external transitions, and NEXT matches the number of external transitions. As a result, compared to the original Johnson method, the size of each element of CHECK became smaller, and the number of elements of the three arrays BASE, CHECK, and NEXT is greatly reduced.

The online construction algorithm [6] is applied to this data structure, because the BASE's index corresponds to the node number.

	0	1	2	3	4	5	6	7
BASE	0	2	3	4				
CHECK		a	b	b	b	a	a	b
NEXT			2	6	5	5		

Figure 6. Triple-array of the proposed method

3.4. The Search Algorithm

Figure 7 shows the algorithm that carries out the search concerning the data structure explained in Section 3.3. We consider that the input of this search is the search string *str*(length *len*), and each character of *str* is stored in *str*[0] to *str*[*len* - 1]. Also, the function CODE returns the internal representation value of a character. This function returns YES if the search succeeds and NO if it fails.

```
Function search(str, len)
(S- 1): s = 0;
(S- 2): for i=0 to len-1 {
(S- 3):     if CHECK[s+1] == str[i] {
(S- 4):         t=s+1;
(S- 5):     }else{
(S- 6):         pos = BASE[s]+CODE(str[i]);
(S- 7):         t = NEXT[pos];
(S- 8):         if CHECK[t] != str[i] {
(S- 9):             return NO;
(S-10):         }
(S-11):     }
(S-12):     s=t;
(S-13): }
(S-14): return YES;
```

Figure 7. The search algorithm of the proposed method

This algorithm first searches for internal transitions, and if the search fails, it searches for external transitions. This is repeated from the beginning of *str* to its last character. In (S-3), it verifies whether a transition can be made from the current node *s* to its following node (*s* + 1) by internal transition. If it is confirmed, the internal transition succeeds and the algorithm proceeds to the next node. If it is not confirmed, it verifies the next external transitions in (S-6) to (S-10). The algorithm acquires the following node *t* from NEXT, and if the transition to that node (CHECK[*t*]) matches the transition character, the transition by external transition is succeeded. If this transition fails, then NO is returned. This process is repeated for all the characters in *str*, and if the transitions are successful for all characters, YES is returned at (S-14) in the end. For example, in Figure 6, when transitioning from node number 3 by 'a', CHECK[4(= 3 + 1)] is checked first. Because CHECK[4] is not 'a', this transition is not the internal transition. Next, the external transition is checked. The next destination node is determined by BASE[3] + 'a' = 3 + 1 = 4 and NEXT[4] = 5, and lastly, by verifying that the value matches the transition character using CHECK[5] = 'a', we can judge that the transition by 'a' from node 3 to node 5 is correct.

4. Experiments and Evaluations

4.1. Experimental Settings

The proposed method and the conventional approaches Table and Johnson method were implemented using C language/C ++. The verification experiments were conducted in the following environment. The machine used was HP ENVY 23, the CPU was Intel (R) Core i7, the memory was 8.00GB, and the OS was Microsoft Windows 10 Pro. Compiling was performed with Visual Studio 2015. As a corpus, we used 50MB of DNA data available for the public on the Pizza & Chili Corpus[1]. For the patterns used in the experiment, we extracted {100,000, 1,000,000} patterns of {10, 50, 100} characters randomly chosen from the corpus.

We evaluated the following items for Table, the Johnson method, and the proposed method.

- Memory capacity required for the representation of the factor oracle.
- Search time of all patterns. This search time will be set as the average time of five different tests.

In the tables of results, TAFO (Triple-Array for Factor Oracle) means the proposed method.

4.2. Experimental Settings

Table 1 shows the experiment results for memory capacity by corpus sizes. Node numbers are represented by 4 bytes. Therefore, each value of Table is represented as 4 bytes, and the BASE values, CHECK values, and NEXT values for the Johnson method are represented as 4 bytes. In the proposed method, the BASE and NEXT values are represented as 4 bytes, and the CHECK values are represented as 1 byte. As shown in Table 1, compared with Table, we found that the proposed method can compress the memory capacity to 4.7%(= 2,458/52,494) at 200KB. Also, compared to the Johnson method, we found that it can compress the memory capacity to 75.1%(= 2,458/3,272) at 200KB.

Table 1. Memory capacity

	1KB	5KB	10KB	30KB	50KB	200KB
Table (KB)	328	1,376	2,687	7,930	13,173	52,494
Johnson method (KB)	17	87	171	502	832	3,272
TAFO (KB)	5	61	123	369	614	2,458
Compared with Table	1.5%	4.4%	4.6%	4.7%	4.7%	4.7%
Compared with Johnson method	29.4%	70.1%	72.0%	73.5%	73.8%	75.1%

Tables 2 to 7 show experimental results for search time by corpus sizes. With the proposed method, compared with Table, we found that the search time is shortened except for the case of 1KB in Table 2. Moreover, compared to the Johnson method, we found that the search time is shortened except for Tables 2 and 3.

Table 2. Search time (milliseconds) when retrieving 100,000 patterns of 10-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	35.2	39.6	40.8	43.8	46.8	48.8
Johnson method	27.6	29.6	30.4	31.2	32.0	33.0
TAFO	36.0	36.8	37.2	39.0	39.8	42.6

Table 3. Search time (milliseconds) when retrieving 1,000,000 patterns of 10-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	360.2	386.2	404.4	420.4	431.8	487.2
Johnson method	281.2	290.8	300.4	308.4	311.0	318.0
TAFO	356.8	367.8	375.6	390.0	395.6	416.2

Table 4. Search time (milliseconds) when retrieving 100,000 patterns of 50-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	114.5	133.6	144.8	157.0	166.6	343.6
Johnson method	98.8	108.8	115.4	144.8	146.4	227.6
TAFO	98.0	99.4	99.6	102.8	103.4	107.0

[1] Pizza & Chili Corpus, http://pizzachili.dcc.uchile.cl/

Table 5. Search time (milliseconds) when retrieving 1,000,000 patterns of 50-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	1,128.4	1,350.4	1,476.4	1,511.4	1,608.4	3,443.8
Johnson method	983.6	1,102.4	1,152.4	1,378.6	1,446.6	1,638.4
TAFO	976.0	992.0	1,000.0	1,017.0	1,029.0	1,068.0

Table 6. Search time (milliseconds) when retrieving 100,000 patterns of 100-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	210.2	245.6	262.8	280.6	294.0	749.6
Johnson method	179.2	206.4	218.4	268.4	283.6	322.0
TAFO	173.2	175.0	176.2	179.2	179.2	184.4

Table 7. Search time (milliseconds) when retrieving 1,000,000 patterns of 100-characters

	1KB	5KB	10KB	30KB	50KB	200KB
Table	2,060.8	2,554.0	2,754.6	2,783.4	3,090.8	7,767.0
Johnson method	1,770.2	2,064.2	2,174.4	2,594.0	2,949.0	3,189.0
TAFO	1,740.6	1,744.4	1,760.8	1,783.2	1,791.8	1,850.2

Tables 8 shows internal transition rates. This rate indicates the ratio of the number of internal transitions examined out of the total number of transitions examined when searching. It is found that as the number of characters in the pattern to be searched increases, the more often internal transitions are examined, and the internal transition rate decreases as the corpus size increases. This is because external transitions increase proportionally as the corpus size augments. As the size of the text increases, there is a higher chance that part of a pattern is present prior to the location where the pattern was acquired, and consecutively the number of external transitions would increase.

Table 8. Internal transition rates (%) for pattern retrieval

	1KB	5KB	10KB	30KB	50KB	200KB
10-characters	56.7	47.4	43.5	37.7	35.2	30.7
50-characters	89.0	88.9	88.1	86.7	86.1	84.4
100-characters	94.9	94.4	94.1	93.3	93.0	92.2

5. Discussion

In Table 1, as far as the storage capacity required for representing the factor oracle, the proposed method carries out the compression efficiently under all conditions. In addition, it was found that the smaller the corpus size, the larger the compressed size. This is because the smaller the size of the corpus, the more the internal transitions exceed the external transitions.

In Tables 2 to 7, when compared with Table, the proposed method is faster except for the case of 1KB in Table 2. This is because in the case of Table, it is likely that cache misses occur more frequently as the memory capacity increases. When compared with the Johnson method, the search time for 10-characters patterns is slower with the proposed method. However, it is faster for 50-characters and 100-characters patterns. Therefore, we chose to focus on the internal transition rate. From Table 8, the 10-characters case had a low internal transition rate of less than 60%, and the 50- and 100-characters cases had high rates of over 80%. Next, we will look at the Johnson method and the search algorithm of the proposed method that was mentioned in Section 2 and Section 3. From Equation (1), the Johnson method needs to confirm the three equations for any transition regardless of whether it is internal or external. In contrast, from the search algorithm in Figure 7, the proposed method first checks in (S-3) the internal transitions using Equation (1), and if it fails then it needs to examine the external transitions in (S-6) to (S-10). In other words, it is considered that the higher the internal transition rate, the faster the proposed method is compared to the Johnson method. In the case of low internal transition rates, it is possible to examine the external transitions first by swapping (S-3) to (S-4) with (S-6) to (S-9) in the search algorithm, by which we can expect a speed improvement.

6. Conclusions

In this paper, we stated the general outline of the factor oracle and the Johnson method and proposed a method aiming to compress the factor oracle using a modified Johnson method.

As far as the conventional methods, we introduced Table and the Johnson method and described their respective characteristics and disadvantages. Table is considered unsuitable for representing automata of large-scale texts because the larger the text, the more memory goes to waste. On the other hand, the Johnson method can be put to use using a transition

speed of O(1), and, like Table, it is a fast method. In addition, the Johnson method has a high space efficiency compared to Table, which makes it an outstanding high-speed method with excellent compactness. In this research, we improved the Johnson method and proposed a method to represent the factor oracle with higher storage efficiency. Moreover, verification experiments showed that the proposed method has higher storage efficiency compared to the Johnson method and is capable under certain conditions of high-speed search. The future task regarding this research consists of increasing the storage efficiency by using succinct data structure.

References

1. P. Weiner, "Linear Pattern Matching Algorithms," in *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1-11, 1973
2. C. Allauzen, M. Crochemore, and M. Raffinot, "Factor Oracle: A New Structure for Pattern Matching," in *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics*, pp. 291- 306, 1999
3. S. C. Johnson, "YACC: Yet Another Compiler Compiler," Bell Laboratories, Murray Hill, NJ, 1978
4. A. V. Aho, R. Sethi, J. D. Ullman, "Compilers-Principles Techniques and Tools," Addison-Wesley, 1986
5. M. Fuketa, K. Morita, and J. Aoe, "Comparisons of Efficient Implementations for DAWG," *International Journal of Computer Theory and Engineering*, Vol. 8, No. 1, pp. 48-52, 2016
6. R. Baeza-Yates and B. Ribeiro-Neto, "Modern Information Retrieval," 2nd Edition, Addison Wesley, 2011
7. D. E. Knuth, J. Morris Jr, and V. Pratt, "Fast Pattern Matching in Strings," *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323-350, 1977
8. A. Maeda and K. Mizushima, "A Compressed-Array Representation of Automata and its Application to Programming Language," in *Proceedings of the 49th Programming Symposium on Information Processing Society of Japan*, pp. 49-54, 2008
9. M. Toro, "Probabilistic Extension to the Concurrent Constraint Factor Oracle Model for Music Improvisation," *Inteligencia Artificial*, Vol. 19, No. 57, pp. 37-73, 2016
10. J. Aoe, "An Efficient Digital Search Algorithm by using a Double-Array Structure," *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066-1077, 1989
11. H. Bast, C.W. Mortensen, and I. Weber, "Output-Sensitive Autocompletion Search," *Information Retrieval*, Vol. 11, No. 4, pp. 269-286, 2008
12. D. E. Knuth, "The Art of Computer Programming, 3," Sorting and Searching, 2nd Edition, Addison Wesley, 1998
13. G. Assayag and S. Dubnov, "Using Factor Oracles for Machine Improvisation," *Soft Computing*, Vol. 8, No. 9, pp. 604-610, 2004
14. S. Yata, K. Morita, M. Fuketa, and J. Aoe, "Fast String Matching with Space-Efficient Word Graphs," in *Proceedings of 2008 International Conference on Innovations in Information Technology*, pp. 79-83, 2008

Koji Bando received his B.E. degree from Tokushima University, Tokushima, Japan, in 1977. After graduating in 1977, He joined NTT Corporation, and he is currently the President and CEO of NTT Plala Inc., Tokyo, Japan. In 2003, he received Telecommunications Association's IT Business Encouragement Special Award. In 2018, he began participating in a doctoral course at Tokushima University. His research interest includes information retrieval, natural language processing, and artificial intelligence.

Takato Nakano received his B.Sc. degree in information science and intelligent systems from Tokushima University in 2017. He currently works for Nichia Corporation.

Kazuhiro Morita received his 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 was a research assistant from 2000 to 2006 in information science and intelligent systems at Tokushima University. He is currently an associate professor in the Department of Information Science and Intelligent Systems at Tokushima University. 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 Tokushima University in 1993, 1995, and 1998, respectively. He was a research assistant and an associate professor respectively from 1998 to 2000 and 2000 to 2015 in information science and intelligent systems at Tokushima University. He is currently a professor in the Department of Information Science and Intelligent Systems at Tokushima University. 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.