Engineering faster double-array Aho–Corasick automata

Shunsuke Kanda[1], Koichi Akabe[1], and Yusuke Oda[1,2]

[1]LegalForce Research, Japan
[2]Center for Data-driven Science and Artificial Intelligence, Tohoku University, Japan

Abstract

Multiple pattern matching in strings is a fundamental problem in text processing applications such as regular expressions or tokenization. This paper studies efficient implementations of *double-array Aho–Corasick automata (DAACs)*, data structures for quickly performing the multiple pattern matching. The practical performance of DAACs is improved by carefully designing the data structure, and many implementation techniques have been proposed thus far. A problem in DAACs is that their ideas are not aggregated. Since comprehensive descriptions and experimental analyses are unavailable, engineers face difficulties in implementing an efficient DAAC.

In this paper, we review implementation techniques for DAACs and provide a comprehensive description of them. We also propose several new techniques for further improvement. We conduct exhaustive experiments through real-world datasets and reveal the best combination of techniques to achieve a higher performance in DAACs. The best combination is different from those used in the most popular libraries of DAACs, which demonstrates that their performance can be further enhanced. On the basis of our experimental analysis, we developed a new Rust library for fast multiple pattern matching using DAACs, named *Daachorse*, as open-source software at https://github.com/daac-tools/daachorse. Experiments demonstrate that Daachorse outperforms other AC-automaton implementations, indicating its suitability as a fast alternative for multiple pattern matching in many applications.

1 Introduction

Multiple pattern matching in strings is a fundamental problem in text and natural language processing [33]. Given a set of patterns and a text, the goal of this problem is to report all occurrences of patterns in the text. A representative application is regular expressions, which provide a powerful way to express a set of search patterns. Another is tokenization in unsegmented natural languages such as Japanese and Chinese, which partitions a sentence into shorter units called tokens or words. Multiple pattern matching is essential in these applications, and its time efficiency is crucial.

The *Aho–Corasick (AC) algorithm* [1] is a fast solution for multiple pattern matching. It uses an *AC automaton* and performs the matching with $O(n)$ character comparisons, where n is the length of an input text. Despite the theoretical guarantee, the practical performance of the AC algorithm is significantly affected by its internal data structure used to represent the AC automaton [35]. Thus, carefully designing the internal data structure is vital to achieve faster matching.

Double-array AC automata (DAACs) are AC-automaton representations for fast matching. The core component is the *double-array* [2], a data structure to implement transition lookups in an optimal time. Prior experiments [35] demonstrated that DAACs performed the fastest compared to various other representations. Thanks to their time efficiency, double-array structures are used in a wide range of applications such as dictionary lookups [28, 46], compressed string dictionaries [26], tokenization [27, 43], language models [36, 44], classification [52], and search engines [9].

To achieve a higher performance in DAACs, it is essential to carefully design the data structure with regard to the target applications and data characteristics. In the three decades since the original idea of the double-array was proposed by Aoe [2], many implementation techniques have been developed from different points of view, such as scalability

[18, 25, 26], cache efficiency [24, 49], construction speed [32, 37, 47], and specializations [36, 44, 47]. In addition to these academic studies, many open-source libraries have been developed, e.g., [20, 28, 40, 46], which sometimes contain original techniques. The problem is that their ideas are not aggregated. Specifically, the comprehensive descriptions and experimental analyses are unavailable, which makes it difficult for engineers to implement an efficient DAAC.

Our contributions In this paper, we review various implementation techniques in DAACs and provide a comprehensive description of them, including categorization for facilitating comparison (as summarized in Table 2). We also propose several new techniques for further improvement. We provide an exhaustive experimental analysis through real-world datasets and reveal the best combination of implementation techniques. The best combination is different from those used in the most popular libraries of DAACs [20, 40], demonstrating that their performance can be further enhanced.

On the basis of our experimental analysis, we develop a new Rust library for fast multiple pattern matching using DAACs, named *Daachorse*, as open-source software at https://github.com/daac-tools/daachorse under the Apache-2.0 or MIT license. Our experiments demonstrate that Daachorse outperforms other AC-automaton implementations, indicating its suitability as a fast alternative for multiple pattern matching in many applications. Daachorse has been plugged into *Vaporetto* [10], a Japanese tokenizer written in Rust. Vaporetto is a fast implementation of the pointwise prediction method [34, 41, 42] that determines token boundaries using a discriminative model. The core step of this method is feature extraction performed with the AC algorithm. Its running time significantly affects the entire processing time, and the fast pattern matching provided by Daachorse assures the time efficiency of Vaporetto. For example, Daachorse performs tokenization 2.6× faster than other implementations, as demonstrated in this paper.

2 Preliminaries

2.1 Basic definition and notation

A *string* is a finite sequence of characters over a finite integer alphabet $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$. Our strings always start at position zero. The empty string ε is a string of length zero. Given a string P of length $n \geq 1$, $P[i..j)$ denotes the *substring* $P[i], P[i+1], \ldots, P[j-1]$ for $0 \leq i \leq j \leq n$. Specially, $P[0..i)$ is a *prefix* of P, and $P[i..n)$ is a *suffix* of P for $0 \leq i \leq n$. Let $|P| := n$ denote the length of P. The same notation is applied to *arrays*. We denote the cardinality of a set A by $|A|$.

2.2 Multiple pattern matching

Given a set of strings $\mathcal{D} = \{P_1, P_2, \ldots, P_{|\mathcal{D}|}\}$ and a string T, the goal of multiple pattern matching is to report all occurrences $\{(k, i, j) : P_k \in \mathcal{D}, P_k = T[i..j)\}$, where an occurrence (k, i, j) consists of the index k of the matched pattern P_k and the starting and ending positions i, j appearing in T. Throughout this paper, we refer to $\mathcal{D}$ as a *dictionary*, P_k as a *pattern*, and T as a *text*.

Table 1 shows an example of a dictionary $\mathcal{D}$ consisting of six patterns. In all examples throughout this paper, we denote indices of patterns by upper-case letters instead of numbers and integer elements in Σ by lower-case letters. Given the dictionary and text $T[0..6) = \texttt{abacdd}$, the occurrences are $(\texttt{A}, 0, 2)$, $(\texttt{B}, 1, 2)$, $(\texttt{D}, 1, 4)$, and $(\texttt{F}, 4, 6)$.

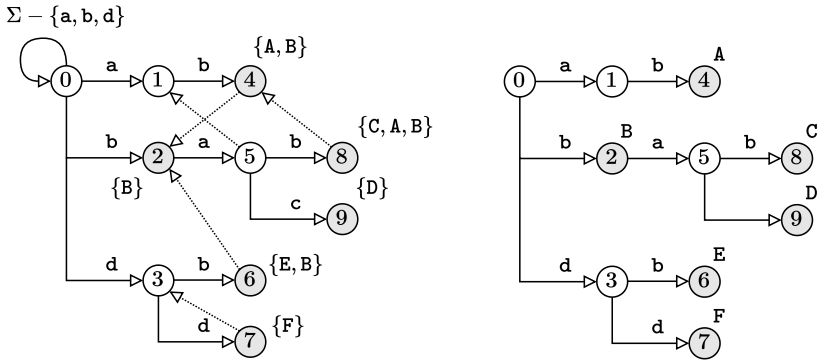
2.3 Aho–Corasick algorithm

The AC automaton [1] is a finite state machine to find all occurrences of patterns in a single scan of a text. The AC automaton for a dictionary $\mathcal{D}$ is defined as the 5-tuple $(S, \Sigma, \delta, f, h)$:

- $S = \{0, 1, \ldots, |S| - 1\}$ is a finite set of states, where each state is identified by an integer, and the initial state is indicated by 0;
- $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$ is the alphabet;

Table 1: A dictionary $\mathcal{D}$ of six patterns (used in examples throughout this paper). Patterns are indexed with upper-case letters $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$ instead of numbers in the examples. Elements in the integer alphabet Σ are denoted with lower-case letters. $\Sigma = \{\mathtt{a} = 0, \mathtt{b} = 1, \mathtt{c} = 2, \mathtt{d} = 3\}$.

Index	Pattern
A	ab
B	b
C	bab
D	bac
E	db
F	dd



(a) AC automaton (b) Trie

Figure 1: Examples of (a) an AC automaton for the dictionary of Table 1 and (b) its trie part. Transitions are depicted by solid line arrows. $\delta(0, \mathtt{b}) = 2$, $\delta(2, \mathtt{a}) = 5$, and $\delta(2, \mathtt{c}) = -1$. We depict the mappings of the failure function (except ones to the initial state) by dotted line arrows. $f(4) = 2$, $f(5) = 1$, $f(6) = 2$, $f(7) = 3$, $f(8) = 4$, and $f(s) = 0$ for the other states s. Output states are shaded and associated with pattern indices (drawn from $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$). $h(2) = \{\mathtt{B}\}$, $h(4) = \{\mathtt{A}, \mathtt{B}\}$, $h(6) = \{\mathtt{E}, \mathtt{B}\}$, $h(7) = \{\mathtt{F}\}$, $h(8) = \{\mathtt{C}, \mathtt{A}, \mathtt{B}\}$, $h(9) = \{\mathtt{D}\}$, and $h(s) = \emptyset$ for the other states s.

- $\delta : S \times \Sigma \to S \cup \{-1\}$ is a transition function, where -1 is an invalid state id;
- $f : S \setminus \{0\} \to S$ is a failure function; and
- $h : S \to \mathcal{P}(\{1, 2, \ldots, |\mathcal{D}|\})$ is an output function, where $\mathcal{P}(\cdot)$ is the power set.

Figure 1a shows an example of the AC automaton.

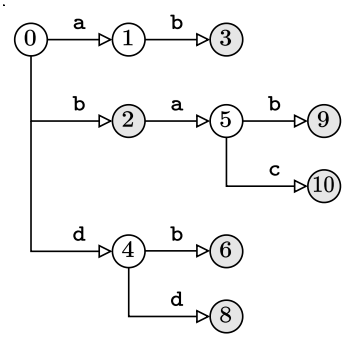
The transition function δ is built on a *trie* for the dictionary $\mathcal{D}$. The trie [16] is a tree automaton formed by merging the prefixes of patterns in $\mathcal{D}$. Figure 1b shows the trie in the AC automaton. A state s in the trie represents any prefix of patterns in $\mathcal{D}$, and the prefix can be extracted by concatenating transition labels from the initial state to state s. We denote by $\phi(s)$ the string represented by state s. For example, $\phi(4) = \mathtt{ab}$ and $\phi(5) = \mathtt{ba}$ in Figure 1b. The initial state always represents the empty string ε, i.e., $\phi(0) = \varepsilon$. We call states satisfying $\phi(s) \in \mathcal{D}$ *output states*. If a state s does not indicate any other state with character c, $\delta(s, c) = -1$ is defined using the invalid state id -1. There is one difference in the definition of δ between the AC automaton and trie: the AC automaton redefines special transitions $\delta(0, c) := 0$ for labels c such that $\delta(0, c) = -1$ in the trie.

The failure function f maps a state $s \in (S \setminus \{0\})$ to another state $t \in (S \setminus \{s\})$ such that $\phi(t)$ is a longer suffix of $\phi(s)$ than $\phi(t')$ for $t' \in (S \setminus \{s, t\})$. Note that, since the empty string ε is a suffix of any string, the initial state is always one of the candidates. Let $F(s)$ be a set of output states reached through only the failure function from state s. For example, $F(8) = \{8, 4, 2\}$ in Figure 1a. The output function $h(s)$ is the set of pattern indices associated with output states in $F(s)$, i.e., $h(s) = \{k : P_k \in \mathcal{D}, P_k = \phi(t), t \in F(s)\}$.

Algorithm 1: Multiple pattern matching using AC automaton.

Input : Text T of length n
Output : All occurrences of patterns $\{(k, i, j) : P_k \in \mathcal{D}, P_k = T[i..j)\}$

```
1 s ← 0
2 for j = 0, 1, ..., n − 1 do
3     for k ∈ h(s) do
4         i ← j − |P_k|
5         Output an occurrence (k, i, j)
6     s ← δ*(s, T[j])
7 for k ∈ h(s) do
8     i ← n − |P_k|
9     Output an occurrence (k, i, n)
```



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

Figure 2: BASE and CHECK implementing the transition function δ of Figure 1b. $\Sigma = \{\mathtt{a} = 0, \mathtt{b} = 1, \mathtt{c} = 2, \mathtt{d} = 3\}$. The state ids are assigned to satisfy Equation (2). $\delta(0, \mathtt{b}) = 2$ is simulated by $\mathsf{BASE}[0] + \mathtt{b} = 2$ and $\mathsf{CHECK}[2] = 0$. $\delta(2, \mathtt{d}) = -1$ is simulated by $\mathsf{BASE}[2] + \mathtt{d} = 8$ and $\mathsf{CHECK}[8] \neq 2$. The state id 7 is a vacant id because its element does not represent any state of the original trie.

Matching algorithm Algorithm 1 shows the AC algorithm that performs multiple pattern matching using the AC automaton. The algorithm uses the extended transition function δ^*:

$$\delta^*(s, c) = \begin{cases} \delta(s, c) & \text{if } \delta(s, c) \neq -1 \\ \delta^*(f(s), c) & \text{otherwise.} \end{cases} \tag{1}$$

Given a text T, the AC algorithm scans T character by character, visits states from the initial state $s = 0$ with δ^*, and reports occurrences where patterns in $h(s)$ are associated with each visited state s. For example, assume we are scanning text $T[0..6) = \mathtt{abacdd}$ with the AC automaton in Figure 1a. We first move with $\mathtt{ab}$ from state 0 to state 4 by $\delta^*(0, \mathtt{a}) = 1$ and $\delta^*(1, \mathtt{b}) = 4$, and then report two occurrences, $(\mathtt{A}, 0, 2)$ and $(\mathtt{B}, 1, 2)$, associated with $h(4)$. Next, we move to state 9 by $\delta^*(4, \mathtt{a}) = 5$ and $\delta^*(5, \mathtt{c}) = 9$ and report occurrence $(\mathtt{D}, 1, 4)$ associated with $h(9)$. Last, we move to state 7 with $\delta^*(9, \mathtt{d}) = 3$ and $\delta^*(3, \mathtt{d}) = 7$ and report occurrence $(\mathtt{F}, 4, 6)$ associated with $h(7)$.

In the scanning, the number of states visited with δ and f is bounded by $2n$ for a text of length n. The algorithm runs in $O(n + occ)$ time in the most efficient case, where occ is the number of occurrences.

2.4 Double-array Aho–Corasick automata (DAACs)

The *double-array* [2] is a data structure to implement the transition function δ using two one-dimensional arrays: *BASE* and *CHECK* (see Figure 2). The double-array arranges original states in S onto BASE and CHECK and assigns new state ids to the original states. An element of BASE and CHECK corresponds to an original state, and its array

Algorithm 2: Construction of BASE and CHECK from an original trie using a queue.

```
Input  : Original trie
Output : BASE and CHECK arrays representing the original trie
Reserve enough elements of BASE and CHECK
Q ← {(0, 0)}                          ▷ Queue to traverse the original trie, initialized with the initial state ids
while Q ≠ ∅ do
    Pop (s, s′) from Q                ▷ s and s′ denote state ids in the original trie and double-array, respectively
    E ← a set of labels of outgoing transitions from state s
    if E ≠ ∅ then
        BASE[s′] ← integer b such that b + c is a vacant id for each c ∈ E      ▷ Vacant search
        for c ∈ E do
            (t, t′) ← (δ(s, c), BASE[s′] + c)
            CHECK[t′] ← s′
            Push (t, t′) to Q
Output BASE and CHECK
```

Algorithm 3: Implementation of $\delta^*(s, c)$ using DAAC.

```
repeat
    t ← BASE[s] + c
    if CHECK[t] = s then
        return t
    else if s = 0 then
        return 0
    s ← FAIL[s]
until
```

offsets indicate new state ids. The BASE and CHECK arrays are constructed so that new state ids satisfy the following equations when $\delta(s, c) = t$ (except for $t = -1$):

$$\mathsf{BASE}[s] + c = t \text{ and } \mathsf{CHECK}[t] = s. \tag{2}$$

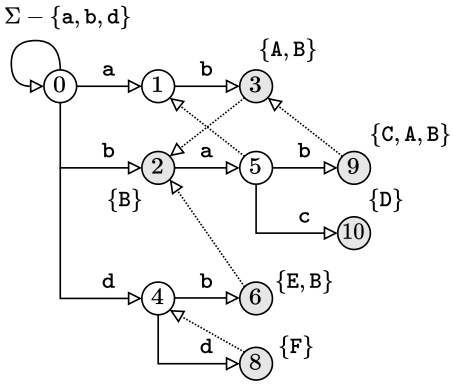
Equation (2) enables us to look up a transition $\delta(s, c)$ in two very simple steps: (i) computing $t := \mathsf{BASE}[s] + c$ and (ii) returning t if $\mathsf{CHECK}[t] = c$ or -1 otherwise. The time complexity is $O(1)$.

Let $S_{\mathsf{BC}} = \{0, 1, \ldots, |S_{\mathsf{BC}}| - 1\}$ be a set of state ids in a resulting double-array structure. The space complexity of BASE and CHECK is $O(|S_{\mathsf{BC}}|)$, where $|S_{\mathsf{BC}}| = |S|$ in the best case and $|S_{\mathsf{BC}}| = |S||\Sigma|$ in the worst case, as S_{BC} can contain unused ids to satisfy Equation (2). For example, state id 7 is unused in Figure 2. We call such state ids *vacant ids*. Constructing BASE and CHECK as few vacant ids as possible is important for space efficiency.

Construction algorithm Constructing BASE and CHECK requires traversing an original trie from the root and searching for values of $\mathsf{BASE}[s]$ for each state s to satisfy Equation (2). More formally, for a state s having outgoing transitions with k labels $c_1, c_2, \ldots, c_k$, we search for a BASE value b such that $b + c_j$ is a vacant id for each $1 \le j \le k$ and define $\mathsf{BASE}[s] = b$ and $\mathsf{CHECK}[b + c_j] = s$, which we call a *vacant search*. Algorithm 2 shows the construction algorithm of BASE and CHECK. In Section 3.5, we will discuss approaches to accelerate vacant searches.

Extension to AC automaton We can extend the double-array to DAACs by introducing components for failure and output functions. Figure 3 shows an example DAAC for the AC automaton in Figure 1a. The failure function f is implemented with array FAIL such that $\mathsf{FAIL}[s] = f(s)$. Using the arrays BASE, CHECK, and FAIL, the extended transition function δ^* is implemented as Algorithm 3.

The output function h is implemented with three arrays: OUTPUT arranges values in $h(s)$ for output states s, TERM stores bit flags to identify terminals of each set $h(s)$ in OUTPUT, and $\mathsf{OUTPOS}[s]$ stores the starting positions of each



	0	1	2	3	4	5	6	7	8	9	10
BASE	1	2	5		5	8					
CHECK		0	0	1	0	2	4		4	5	5
FAIL	0	0	0	2	0	1	2		4	3	0
OUTPOS			0	1			3		5	6	9

	0	1	2	3	4	5	6	7	8	9
OUTPUT	B	A	B	E	B	F	C	A	B	D
TERM	1	0	1	0	1	1	0	0	1	1

Figure 3: DAAC for the AC automaton in Figure 1a.

set $h(s)$ in OUTPUT. Using these arrays, we can extract $h(s)$ by scanning OUTPUT$[i..j]$ from the starting position i = OUTPOS$[s]$ until encountering TERM$[j] = 1$.

3 Implementation techniques in DAACs

In this section, we review implementation techniques to improve the DAAC performance and describe them based on our categorization (summarized in Table 2).

3.1 Management of output sets

We first describe how to manage output sets efficiently. Figure 3 shows a simple approach that arranges values in $h(s)$ in an array. We call this approach Simple. An advantage of Simple is the *locality of reference* that can extract $h(s)$ by a sequential scan. However, Simple can maintain many duplicate values in OUTPUT. For example, value B appears four times in the OUTPUT of Figure 3. The length of OUTPUT is bounded by $O(|\mathcal{D}| \cdot K)$, where K is the average length of patterns in the dictionary. In the following, we present two additional approaches, Shared and Forest, to improve the memory efficiency of Simple.

Shared arrangement To design Shared, we exploit the fact that $h(t) \subseteq h(s)$ when state t can be reached from state s through only failure links. For example, $h(2) \subseteq h(3) \subseteq h(9)$ in Figure 3. This fact indicates that values in $h(t)$ can be represented as a part of the values in $h(s)$. Shared implements the OUTPUT and TERM arrays while merging such common parts. Figure 4a shows an example of Shared, where $h(2)$ and $h(3)$ are represented in the rear parts of OUTPUT$[3..5]$ for $h(9)$. Shared can extract $h(s)$ in the same manner as Simple and does not lose the locality of reference.

Forest representation A drawback of Shared is that it cannot remove all duplicate values, and the space complexity of OUTPUT and TERM is not improved. For example, value B still appears twice in Figure 4a. Forest is an alternative approach that maintains only unique pattern indices [7].

Multiple trees, or a *forest*, can be constructed from an AC automaton by chaining output states through the failure function, which we call an *output forest*. The left part of Figure 4b shows such an output forest constructed from the AC automaton in Figure 3. In the output forest, we can extract values in $h(s)$ by climbing up the corresponding tree to the root.

Forest constructs an output forest whose nodes are indexed by numbers from $\{0, 1, \ldots, |\mathcal{D}| - 1\}$. Our data structure represents the forest using two arrays, OUTPUT and PARENT, such that OUTPUT stores pattern indices and PARENT stores parent positions. Also, OUTPOS$[s]$ stores the node index corresponding to an output state s in the

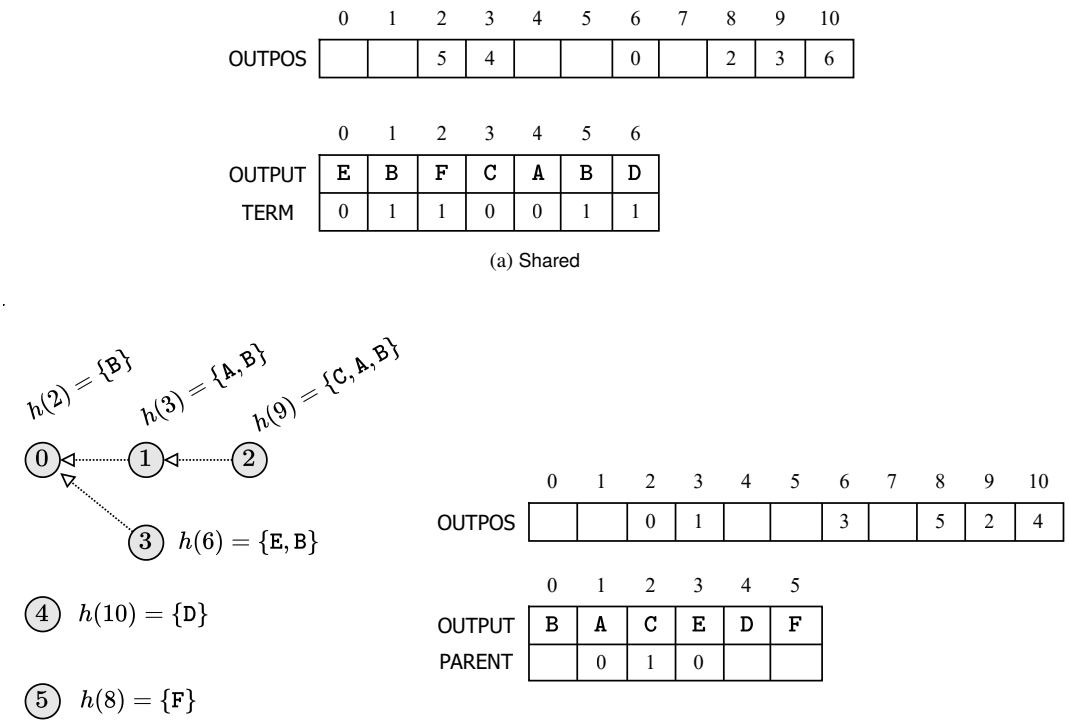


Figure 4: Examples of approaches to store output sets of Figure 3.

AC automaton. $h(s)$ is extracted by visiting OUTPUT$[i]$ from node $i :=$ OUTPOS$[s]$ while updating $i :=$ PARENT$[i]$ until encountering the root. The right part of Figure 4b shows the data structure.

The advantage of Forest is that the space complexity of OUTPUT and PARENT is bounded by $O(|\mathcal{D}|)$. However, traversing an output forest requires random accesses, which sacrifices the locality of reference in Simple and Shared. Also, the pointer array PARENT consumes a larger space than the bit array TERM.

3.2 Byte- and character-wise automata

When handling strings consisting of multibyte characters, there are two representations of strings:

- Bytewise represents strings as sequences of bytes in the UTF-8 format and uses byte values for transition labels. The maximum value in the alphabet Σ is the maximum byte value appearing in input patterns.
- Charwise represents strings as sequences of code points in Unicode and uses code-point values for transition labels. The maximum value in the alphabet Σ is the maximum code-point value appearing in input patterns.

For example, the Japanese string "世界" (meaning "the world") is represented as a sequence of six bytes "0xE4, 0xB8, 0x96, 0xE7, 0x95, 0x8C" in Bytewise and a sequence of two code points "U+4E16, U+754C" in Charwise. The alphabet Σ in Bytewise is $\{0x00, 0x01, \ldots, 0xE7\}$, and the alphabet Σ in Charwise is $\{U+0000, U+0001, \ldots, U+754C\}$.

In the following, we first describe the advantages and disadvantages when constructing DAACs from strings in the Bytewise or Charwise scheme; then, we present an approach called Mapped to overcome the disadvantages in the Charwise scheme.

Advantages and disadvantages There is a trade-off between the alphabet size (or $|\Sigma|$) and the number of automaton states (or $|S|$): the alphabet size in Bytewise is bounded by 2^8 and is smaller than that in Charwise bounded by 2^{21}

(since code points are drawn up to U+10FFFF), and the number of states in Charwise is lower than that in Bytewise. For example, consider an AC automaton for the single pattern "世界". The alphabet size in Bytewise is 232 (i.e., 0xE7 plus 1) while that in Charwise is 30029 (i.e., U+754C plus 1); thus, the number of states in Bytewise is seven while that in Charwise is three.

Smaller alphabets enable DAACs to achieve faster construction because the possible number of outgoing transitions from a state is suppressed, which facilitates vacant searches. Fewer states enable DAACs to achieve faster matching because of suppressing the number of random memory accesses during a matching. Therefore, Bytewise is beneficial for faster construction, and Charwise is beneficial for faster matching.

The memory usage of DAACs depends on the number of resulting double-array states $|S_{\mathsf{BC}}|$, which is the sum of the number of original states $|S|$ and that of vacant ids. Although Charwise can construct an AC automaton with fewer states, its large alphabet can produce more vacant ids because of the difficulty in vacant searches. Thus, we can achieve memory efficiency by constructing a DAAC with fewer vacant ids in the Charwise scheme.

Code mapping The Mapped approach can overcome the disadvantages in Charwise by mapping code points to smaller integers [29]. This approach assigns code values to characters $c \in \Sigma$ with a certain frequency to assign smaller code values to more frequent characters.

We give a formal description of Mapped. Let us denote by $\#(c)$ the number of occurrences of character $c \in \Sigma$ in patterns of $\mathcal{D}$. We construct the *mapping function* $\pi : \Sigma \to \Sigma_\pi$ such that:

- $\Sigma_\pi = \{-1, 0, 1, \ldots, \sigma - 1\}$, where σ is the number of characters c such that $\#(c) \neq 0$;
- $\pi(c) = -1$ for characters c when $\#(c) = 0$; and
- $\pi(c)$ is the number of other characters c' such that $\#(c) < \#(c')$ (breaking ties arbitrarily).

We call σ the *mapped alphabet size*.

Also, Equation (2) is modified into

$$\mathsf{BASE}[s] + \pi(c) = t \text{ and } \mathsf{CHECK}[t] = s. \tag{3}$$

Note that characters c such that $\pi(c) = -1$ are not used in transitions because (i) during construction, the characters do not appear, and (ii) during pattern matching, we can immediately know whether transitions with the characters fail.

If Σ includes many characters c such that $\#(c) = 0$, the mapped alphabet size σ becomes much smaller than $|\Sigma|$. Moreover, occurrences of real-world characters are often skewed following Zipf's law [51], indicating that most of the characters in $\mathcal{D}$ are represented with small code values via the mapping π. Prior works have empirically demonstrated that mapping π reduces the resultant vacant ids and shortens the construction time [29, 36, 44].

However, we need to store an additional data structure for implementing the mapping π. We implement π with an array of length $|\Sigma|$ such that the c-th element stores the value of $\pi(c)$. When the array consists of 2^{21} four-byte integers, it takes 8 MiB of memory.

3.3 Memory layout of arrays

This paper says that the data structure of DAACs consists of four arrays: BASE, CHECK, FAIL, and OUTPOS. In the following, we describe the two memory layouts of the four arrays: Individual and Packed.

Individual layout The Individual layout maintains the four arrays individually, as shown in Figure 5a.

Packed layout Since the values of $\mathsf{BASE}[s]$, $\mathsf{CHECK}[s]$, $\mathsf{FAIL}[s]$, and $\mathsf{OUTPOS}[s]$ are accessed consecutively during a matching, we should be able to improve the locality of reference by placing these values in a consecutive memory array. The Packed layout represents states using a one-dimensional array STATE, where each element consists of the four fields base, check, fail, and outpos corresponding to each array. Figure 5b illustrates the Packed layout.

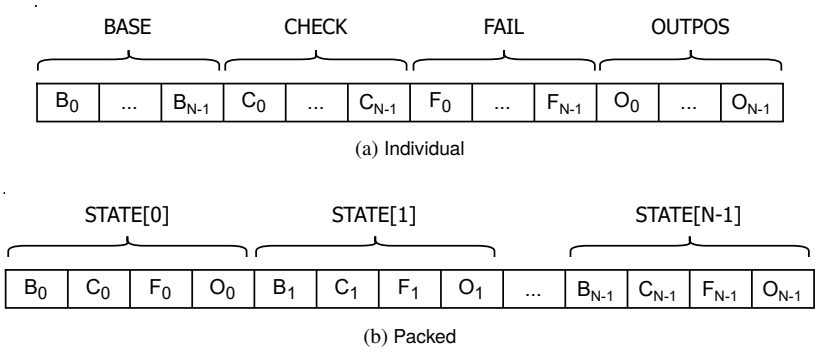


Figure 5: Illustrations of memory layouts of arrays.

3.4 Array formats

Double-array implementations often use 4-byte integers to represent an offset of arrays (e.g., [28, 46]). When BASE, CHECK, FAIL, and OUTPOS are implemented as arrays of 4-byte integers, we call this format Basic. In the following, we describe a more memory-efficient format that implements CHECK as a byte array in the Bytewise scheme, called Compact.

Compact format The Compact format stores transition labels in CHECK instead of array offsets [49]. In other words, CHECK$[t]$ stores c instead of s when $\delta(s, c) = t$, and Equation (2) is modified into

$$\mathsf{BASE}[s] + c = t \text{ and } \mathsf{CHECK}[t] = c. \tag{4}$$

However, for every state pair (s, s'), the following must be satisfied to avoid defining invalid transitions:

$$\mathsf{BASE}[s] \neq \mathsf{BASE}[s']. \tag{5}$$

In the Bytewise scheme, the alphabet size is bounded by 256, and CHECK can be implemented as a byte array. The Compact format can reduce the memory consumption of CHECK to 25% while maintaining the time efficiency in matching. In the Packed layout, we assign three bytes to base, one byte to check, and four bytes to fail and outpos for avoiding extra memory padding on a 4-byte aligned memory structure, following the original implementation [49].[1]

A drawback of Compact is that it makes it harder for vacant searches to avoid the duplication of BASE values for satisfying Equation (5); indeed, this restriction can produce vacant ids that never satisfy Equation (5). For example, a vacant id s never satisfies Equation (5) when BASE values $s, s-1, \ldots, s-|\Sigma|+1$ are already used. In Section 4.5, we observe that such vacant ids can significantly slow the running times of vacant searches.

3.5 Acceleration of vacant searches

Given a state having k outgoing transitions with labels $c_1, c_2, \ldots, c_k$, a naïve vacant search is performed to verify if $b + c_j$ for $1 \leq j \leq k$ are vacant ids for each integer $b \geq 0$ until finding such a value b. However, for a maximum state id N, this approach verifies $O(N)$ integers in the worst case, resulting in slow construction for a large automaton. In the following, we describe three acceleration techniques for vacant searches: Chain, SkipForward, and SkipDense.

Chaining vacant ids The Chain technique constructs a linked list on vacant ids [32]. Let us denote M vacant ids by $q_1, q_2, \ldots, q_M$, where $0 < q_1 < q_2 < \cdots < q_M < N$. Given a state having k outgoing transitions with labels

[1]This modification limits the maximum state id to $2^{24} - 1$. To represent state ids up to $2^{29} - 1$ with the same space usage, we can employ a technique used in the Darts-clone library [46] that utilizes two types of offset values to represent exact and rough positions. We did not use this technique in our experiments because three bytes are enough to store the automata we tested.

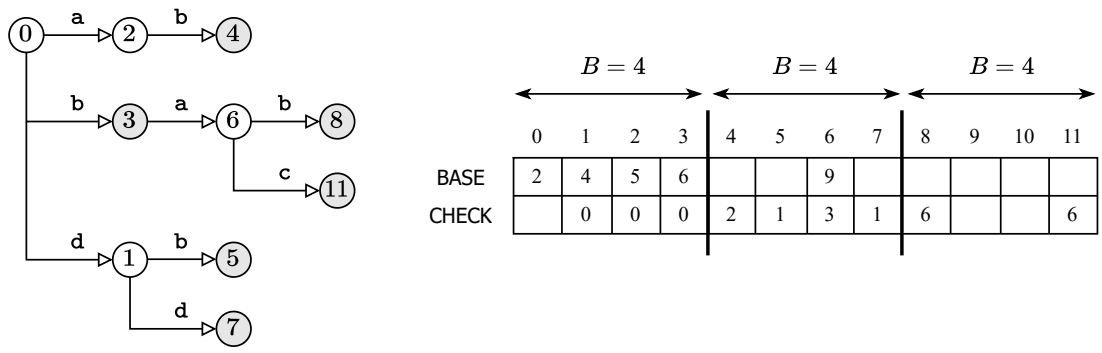


Figure 6: BASE and CHECK implementing the transition function δ of Figure 1b based on Equation (6). The block size B is four, since $|\Sigma| = 4$ and $B = 2^{\lceil \log_2 4 \rceil}$. Destination states 5 and 7 from state 1 are placed in the same block, i.e., $\lfloor 5/4 \rfloor = \lfloor 7/4 \rfloor = 1$.

$c_1, c_2, \ldots, c_k$, Chain verifies only BASE values b_i such that $b_i = q_i - \min\{c_1, c_2, \ldots, c_k\}$ for $i = 1, 2, \ldots, M$ while visiting only vacant ids using the linked list. The number of verifications is bounded by $O(M)$.

If there are few vacant ids, M becomes much smaller than N, and vacant searches can be performed faster than with the naïve approach. The resultant double-array structure with Chain is always identical to that with the naïve approach.

Skipping search blocks Before introducing SkipForward and SkipDense, we present a technique to partition array elements into fixed-size blocks [24, 46, 52]. The partitioning modifies Equation (2) into

$$\mathsf{BASE}[s] \oplus c = t \text{ and } \mathsf{CHECK}[t] = s. \tag{6}$$

In other words, the bitwise-XOR operation ($\oplus$) is used instead of the plus operation (+).

We introduce a constant parameter B and partition array elements into blocks of size B, where the s-th element is placed in the $\lfloor s/B \rfloor$-th block. We obtain the following theorem.

Theorem 1. *Let $B = 2^{\lceil \log_2 |\Sigma| \rceil}$.*[2] *When state ids are defined using Equation (6), all destination states from a state are always placed in the same block.*

Proof. Computing $\mathsf{BASE}[s] \oplus c$ is implemented by modifying the least significant $\lceil \log_2 |\Sigma| \rceil$ bits of $\mathsf{BASE}[s]$. For all characters $c \in \Sigma$, the results of $\mathsf{BASE}[s] \oplus c$ have the same binary representation except the least significant $\lceil \log_2 |\Sigma| \rceil$ bits. Computing $\lfloor x/B \rfloor$ for an integer x is implemented by shifting $\lceil \log_2 B \rceil = \lceil \log_2 |\Sigma| \rceil$ bits right, and the least significant $\lceil \log_2 |\Sigma| \rceil$ bits of x are omitted from the result. Therefore, $\lfloor (\mathsf{BASE}[s] \oplus c)/B \rfloor = \lfloor \mathsf{BASE}[s]/B \rfloor$ for all characters $c \in \Sigma$, and all destination states t from a state s are placed in the $\lfloor \mathsf{BASE}[s]/B \rfloor$-th block. □

Figure 6 shows an example of BASE and CHECK constructed using Equation (6). The partitioning allows us to perform vacant searches for blocks individually and set up conditions if we search each block. SkipForward and SkipDense each use a different approach to select which blocks to be searched.

SkipForward searches for only the last L blocks with a constant parameter L [46], as shown in Figure 7a. This idea is inspired by the fact that backward state ids are more likely to be vacant when performing vacant searches from the forward [32]. SkipForward can bound the number of verifications in a vacant search by $O(L \cdot |\Sigma|) = O(|\Sigma|)$, since L is constant. A drawback of SkipForward is that blocks containing many vacant ids might be skipped, which degrades the memory efficiency.

SkipDense introduces a constant threshold $\tau \in [0, 1]$ and categorizes blocks into two classes: if the proportion of vacant ids in a block is no less than τ, the block is categorized to the *sparse* class; otherwise, the block is categorized to the *dense* class. SkipDense performs vacant searches for only sparse blocks, as shown in Figure 7b. SkipDense

[2] In the Mapped scheme, σ is used instead of $|\Sigma|$.

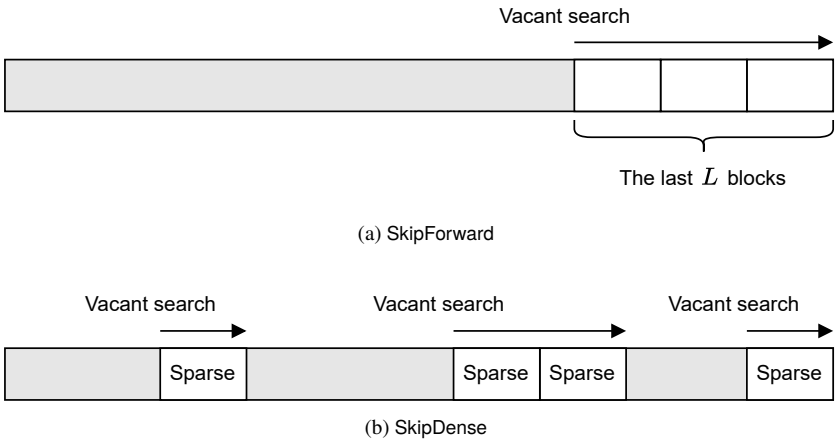


Figure 7: Illustrations of skipping techniques.

does not skip blocks that contain many vacant ids and is thus more memory-efficient than SkipForward. However, the number of verifications is not bounded by the alphabet size.[3]

The ideas of both SkipForward and SkipDense do not conflict with Chain. We therefore apply the technique of Chain to SkipForward and SkipDense.

3.6 Traversal orders in construction

BASE and CHECK are constructed by traversing an original trie from the root and performing vacant searches, as shown in Algorithm 2. Although this algorithm traverses the original trie using a queue in the breadth-first order, we can take an arbitrary order to traverse the trie. We can also take an arbitrary scan order of outgoing transitions for each visited state (i.e., the loop order of E at Line 8 in Algorithm 2).

The orders are related to the resultant arrangement of states in BASE and CHECK, and the arrangement affects the cache efficiency in transitions: a transition $\delta(s, c) = t$ can be looked up quickly when states s and t are placed close together (e.g., on the same cache line). Selecting the traversal order that improves the locality of reference during a matching is crucial. In the following, we describe the four approaches: LexBFS, LexDFS, FreqBFS, and FreqDFS.

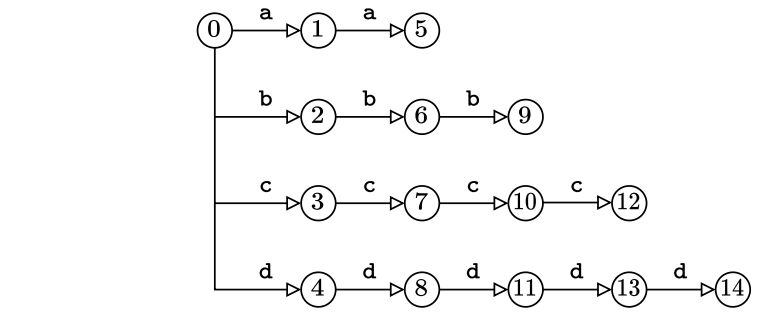
Breadth- and depth-first searches There are two types of data structures to traverse the original trie. One uses a queue and visits states with a breadth-first search (as Algorithm 2), which we denote by BFS. The other uses a stack and visits states with a depth-first search, which we denote by DFS.

We consider performing vacant searches from the forward (i.e., attempting to use smaller state ids first). The resultant arrangements of states using BFS and DFS are expected to be as follows. With BFS, it is expected that shallower states (i.e., ones around the initial state) are placed around the head of BASE and CHECK. If we often visit shallow states during a matching, BFS can perform cache-efficiently. With DFS, in contrast, it is expected that deeper states are placed close together. If we often visit deep states during a matching, DFS can perform cache-efficiently.

Figure 8 shows toy examples of resultant BASE and CHECK arrays with BFS and DFS. In BFS, shallower states have smaller state ids (see Figure 8a), while in DFS, states whose depth is no less than two have adjacent ids with their destination states (see Figure 8b). These examples show that BFS and DFS have better locality of reference when we visit shallower and deeper states, respectively.

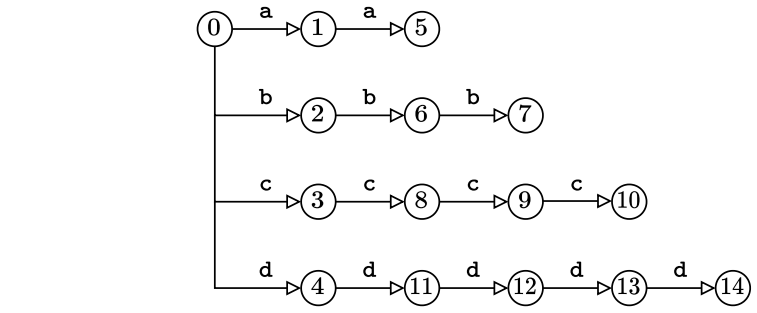
Lexicographical and frequency orders There are two orders to traverse outgoing transitions for each visited state: Lex visits transitions in the lexicographical order of their labels, and Freq visits transitions in the decreasing order

[3] SkipDense is inspired by the idea in the Darts library [28], which skips forward elements in which the proportion of vacant ids is not less than a certain threshold. SkipDense is a minor modification of the Darts technique combined with block partitioning and enables vacant searches to select target blocks more flexibly.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
BASE	1	5	5	5	5		8	8	8		10	10		11	
CHECK		0	0	0	0	1	2	3	4	6	7	8	10	11	13

(a) BFS



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
BASE	1	5	5	6	8		6		7	8		9	10	11	
CHECK		0	0	0	0	1	2	6	3	8	9	4	11	12	13

(b) DFS

Figure 8: Examples of resultant BASE and CHECK arrays constructed with BFS and DFS for a dictionary of four patterns aa, bbb, cccc, and ddddd. $\Sigma = \{\mathtt{a} = 0, \mathtt{b} = 1, \mathtt{c} = 2, \mathtt{d} = 3\}$. The state ids are determined based on Equation (2). In the construction, vacant searches are performed from the forward to use smaller state ids first. A set of outgoing transition labels (or E) is scanned in the lexicographical order (i.e., the figures show the results of LexBFS and LexDFS more precisely). In both cases, the initial state and its destination states have identical ids (i.e., states 0 to 4). This is because the initial state id is always fixed to 0, and its destination state ids are fixed to BASE[0] to satisfy Equation (2). The deeper states have different ids in accordance with the order of visiting states.

Table 2: Summary of implementation techniques in DAACs. "Selected" indicates the techniques selected to be the best through our experiments in Section 4 and are used in our Daachorse library.

(a) Approaches to store output sets (Section 3.1)

Technique	Brief description	References	Selected
Simple	Arranging pattern indices in a simple manner	Conventional	
Shared	Merging some common parts of pattern indices in Simple	This study	
Forest	Storing pattern indices in a forest structure	[7]	✓

(b) Schemes to handle strings of multibyte characters (Section 3.2)

Technique	Brief description	References	Selected
Bytewise	Slicing multibyte characters into byte sequences	Conventional	✓
Charwise	Handling multibyte characters as code points in Unicode	Conventional	
Mapped	Mapping code points in the frequency order	[29, 36]	✓

(c) Memory layouts of double-arrays (Section 3.3)

Technique	Brief description	References	Selected
Individual	Maintaining arrays BASE, CHECK, FAIL, and OUTPOS individually	Conventional	
Packed	Arranging values in the arrays cache-efficiently	e.g., [32, 49]	✓

(d) Formats of double-arrays in Bytewise scheme (Section 3.4)

Technique	Brief description	References	Selected
Basic	Implementing BASE, CHECK, FAIL, and OUTPOS as arrays of four-byte integers	Conventional	
Compact	Compressing CHECK into a byte array	[49]	✓

(e) Approaches to accelerate vacant searches (Section 3.5)

Technique	Brief description	References	Selected
Chain	Visiting only vacant ids with a linked list	[32]	
SkipForward	Searching for only the last L blocks	[32, 46]	✓
SkipDense	Searching for only blocks in which the vacant proportion is no less than τ	This study	

(f) Traversal orders in construction (Section 3.6)

Technique	Brief description	References	Selected
LexBFS	Visiting states with breadth-first search and transition labels in the lexicographical order	Conventional	
FreqBFS	Visiting states with breadth-first search and transition labels in the frequency order	Conventional	
LexDFS	Visiting states with depth-first search and transition labels in the lexicographical order	Conventional	✓
FreqDFS	Visiting states with depth-first search and transition labels in the frequency order	Conventional	

of the frequencies of their labels in $\mathcal{D}$. In the Freq order, ids of states incoming with frequent transition labels are determined earlier, and it is expected that those states are placed at the head of the array.

Possible combinations There are four possible combinations in the traversal order:

- LexBFS is BFS with Lex,
- FreqBFS is BFS with Freq,
- LexDFS is DFS with Lex, and
- FreqDFS is DFS with Freq.

3.7 Summary of implementation techniques

Table 2 summarizes the implementation techniques described in this section.

Table 3: Basic statistics of dictionaries.

(a) Average pattern length in bytes

Dataset	1K	10K	100K	1M
EnWord	5.0	6.6	7.1	7.5
JaWord	5.7	6.7	8.7	14.5
JaChars	4.9	6.3	7.3	8.0

(b) Average pattern length in characters

Dataset	1K	10K	100K	1M
EnWord	5.0	6.6	7.1	7.5
JaWord	1.9	2.3	3.0	5.2
JaChars	1.8	2.3	2.7	2.9

(c) Alphabet size in the Bytewise scheme

Dataset	1K	10K	100K	1M
EnWord	226	226	239	239
JaWord	233	238	239	244
JaChars	233	233	238	239

(d) Alphabet size in the Charwise scheme

Dataset	1K	10K	100K	1M
EnWord	8,226	9,632	63,289	65,532
JaWord	39,640	57,345	64,017	1,048,766
JaChars	39,640	40,845	57,345	64,017

(e) Mapped alphabet size in the Mapped scheme

Dataset	1K	10K	100K	1M
EnWord	83	104	177	472
JaWord	669	2,262	4,806	10,809
JaChars	483	1,648	2,990	4,773

Table 4: The number of occurrences in pattern matching per sentence.

Dataset	1K	10K	100K	1M
EnWord	47.2	93.4	126.2	150.8
JaWord	25.8	38.0	46.7	49.0
JaChars	36.6	54.6	69.7	79.9

4 Experimental analyses of DAAC techniques

4.1 Setup

We conducted all experiments on one core of a hexa-core Intel i7-8086K CPU clocked at 2.80 GHz in a machine with 64 GB of RAM (L1 cache: 32 KiB, L2 cache: 256 KiB, L3 cache: 12 MiB), running the 64-bit version of CentOS 7.5 based on Linux 3.10. All data structures were implemented in Rust. We compiled the source code by rustc 1.60.0 with optimization flag `opt-level=3`.

Datasets We use three real-world natural language datasets:

- EnWord is English word uni-grams in the Google Web 1T 5-Gram Corpus [8];
- JaWord is Japanese word uni-grams in the Nihongo Web Corpus 2010 [45]; and
- JaChars is Japanese character uni-, bi-, and tri-grams in the Nihongo Web Corpus 2010 [45].

In these datasets, each N-gram has its own frequency. We extract the most frequent K N-grams from each dataset and produce dictionaries $\mathcal{D}$ of K patterns. We test $K = 10^3, 10^4, 10^5, 10^6$ to observe scalability. Table 3 lists the basic statistics of the dictionaries.

We evaluate the running times of pattern matching and the construction and memory usage of an AC automaton. We use an English text from the Pizza&Chili Corpus [15] and a Japanese text from 13 different subcorpora in the Balanced Corpus of Contemporary Written Japanese (BCCWJ, version 1.1) [30] to measure pattern matching times for the English and Japanese dictionaries, respectively. We randomly sample one million sentences (separated by lines) from each text and produce a sequence of search texts T. Table 4 shows the number of occurrences during a pattern matching for each dataset. Throughout the experiments, we report the total running time of a pattern matching for all sentences, averaged on ten runs.

4.2 Analysis on approaches to store output sets

Table 5 shows the experimental results on the Simple, Shared, and Forest approaches presented in Section 3.1, while fixing the other settings to Bytewise, Packed, Basic, Chain, and LexDFS.

Table 5: Experimental results for approaches to store output sets: Simple, Shared, and Forest. For memory usage, the total of OUTPUT and TERM is reported in Simple and Shared, and the total of OUTPUT and PARENT is reported in Forest. The last two rows in Tables (a)–(c) show the different ratios for each result.

(a) Length of OUTPUT ($\times 10^3$)

	EnWord				JaWord				JaChars			
Approach	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Simple	1.60	29.2	412	4,988	1.46	18.7	247	3,615	1.68	21.6	252	2,779
Shared	1.46	26.2	362	4,377	1.31	16.7	218	3,132	1.40	17.7	217	2,497
Forest	1.00	10.0	100	1,000	1.00	10.0	100	1,000	1.00	10.0	100	1,000
Shared/Simple	0.92	0.90	0.88	0.88	0.89	0.89	0.88	0.87	0.83	0.82	0.86	0.90
Forest/Simple	0.63	0.34	0.24	0.20	0.68	0.53	0.40	0.28	0.60	0.46	0.40	0.36

(b) Memory usage (KiB)

	EnWord				JaWord				JaChars			
Approach	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Simple	12.5	228	3,219	38,968	11.4	146	1,932	28,244	13.1	168	1,971	21,707
Shared	11.4	205	2,830	34,192	10.2	131	1,703	24,473	10.9	139	1,692	19,507
Forest	11.7	117	1,172	11,719	11.7	117	1,172	11,719	11.7	117	1,172	11,719
Shared/Simple	0.92	0.90	0.88	0.88	0.89	0.89	0.88	0.87	0.83	0.82	0.86	0.90
Forest/Simple	0.94	0.51	0.36	0.30	1.02	0.80	0.61	0.41	0.89	0.70	0.59	0.54

(c) Matching time (ms)

	EnWord				JaWord				JaChars			
Approach	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Simple	525	609	693	1,088	469	646	919	1,266	448	700	1,073	2,853
Shared	525	613	719	1,148	457	653	923	1,304	415	668	1,044	2,822
Forest	533	596	697	1,088	446	626	871	1,259	427	663	1,027	2,796
Shared/Simple	1.00	1.01	1.04	1.06	0.98	1.01	1.00	1.03	0.92	0.95	0.97	0.99
Forest/Simple	1.01	0.98	1.00	1.00	0.95	0.97	0.95	0.99	0.95	0.95	0.96	0.98

(d) Average cardinality of $h(s)$

Dataset	1K	10K	100K	1M
EnWord	1.37	2.07	3.16	4.15
JaWord	1.42	1.76	2.20	3.02
JaChars	1.63	2.06	2.42	2.73

Table 6: Experimental results on memory efficiency in the Bytewise, Charwise, and Mapped schemes. The memory usage is the total of BASE, CHECK, FAIL, and OUTPOS (and the mapping π in Mapped). The last rows in Tables (a) and (d) show the different ratios for each result.

(a) Number of original states ($\times 10^3$)

	EnWord				JaWord				JaChars			
Scheme	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	2.74	25.6	237	2,129	2.96	27.9	305	4,899	1.73	18.5	186	1,790
Charwise	2.74	25.6	237	2,124	1.43	13.0	142	2,155	1.12	10.9	107	1,052
Charwise/Bytewise	1.00	1.00	1.00	1.00	0.48	0.47	0.46	0.44	0.65	0.59	0.58	0.59

(b) Proportion of vacant ids (%)

	EnWord				JaWord				JaChars			
Scheme	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	2.6	0.9	0.1	0.0	3.7	0.8	0.2	0.1	15.3	0.8	0.2	0.0
Charwise	66.6	0.2	0.0	0.0	96.4	77.3	16.3	0.7	97.2	84.1	68.0	71.8
Mapped	2.7	0.9	0.2	0.0	30.2	20.5	3.9	1.1	27.1	24.1	12.6	27.0

(c) Average number of outgoing transitions for an internal state.

	EnWord				JaWord				JaChars			
Scheme	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	1.41	1.40	1.44	1.50	1.41	1.43	1.37	1.20	1.87	1.68	1.75	1.91
Charwise	1.41	1.41	1.44	1.50	2.54	2.79	2.40	1.60	3.58	3.25	3.83	5.33

(d) Memory usage (KiB)

	EnWord				JaWord				JaChars			
Scheme	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	44	404	3,712	33,276	48	440	4,780	76,596	32	292	2,908	27,972
Charwise	128	401	3,707	33,193	619	896	2,646	33,922	619	1,067	5,240	58,318
Mapped	76	442	3,959	33,456	187	480	2,554	38,145	179	384	2,144	22,778
Charwise/Bytewise	2.91	0.99	1.00	1.00	12.89	2.04	0.55	0.44	19.34	3.65	1.80	2.08
Mapped/Bytewise	1.73	1.09	1.07	1.01	3.89	1.09	0.53	0.50	5.59	1.31	0.74	0.81
Mapped/Charwise	0.59	1.10	1.07	1.01	0.30	0.54	0.97	1.12	0.29	0.36	0.41	0.39

We first focus on the statistics related to memory efficiency. Table 5a shows the length of OUTPUT, i.e., the number of pattern indices stored in the data structure. The length in Forest is essentially the same as the number of patterns and is much smaller than that in Simple, indicating that Simple maintains many duplicate values in OUTPUT. Shared also reduces such duplicate values in Simple. Comparing Shared and Forest on their reduction ratios from Simple, the difference becomes larger as the number of patterns increases, and Forest is more memory efficient for large dictionaries. Table 5b shows the memory usage of the data structures. Shared is the smallest when the number of patterns is 1K, and Forest is the smallest in the other cases.

We next focus on the statistics related to the time efficiency of the AC algorithm. Table 5c reports the elapsed time during a pattern matching, and as we can see, there is no significant difference between the approaches. These results indicate that the cache-efficient scanning in Simple and Shared is unimportant. To clarify this, Table 5d shows the average cardinality of $h(s)$ for each dictionary. These results imply that only a few random accesses can arise in Forest's extraction and do not create a bottleneck.

4.3 Analysis on byte-wise and character-wise schemes

We evaluate the performances of the Bytewise, Charwise, and Mapped schemes presented in Section 3.2, while fixing the other settings to Forest, Packed, Basic, Chain, and LexDFS.

Table 6 shows the results on memory efficiency, where Table 6a lists the number of original states (i.e., $|S|$) and Table 6b reports the proportion of vacant ids (i.e., $|S|/|S_{\mathsf{BC}}|$), which we call *vacant proportion*. On JaWord and

Table 7: Experimental results on time efficiency in the Bytewise, Charwise, and Mapped schemes. The last rows in Tables (a) and (c) show the different ratios for each result.

(a) Number of visiting states during matching ($\times 10^6$)

Scheme	EnWord				JaWord				JaChars			
	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	108	104	107	111	128	128	129	129	129	129	129	131
Charwise	108	104	107	111	57	61	61	61	59	62	62	62
Charwise/Bytewise	1.00	1.00	1.00	1.00	0.44	0.48	0.47	0.47	0.45	0.48	0.48	0.48

(b) Matching time (ms)

Scheme	EnWord				JaWord				JaChars			
	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	527	597	689	1,057	454	644	896	1,254	406	648	1,021	2,735
Charwise	550	628	720	1,098	362	444	563	809	356	507	661	1,812
Mapped	563	654	741	1,135	315	407	533	762	304	424	625	1,677
Charwise/Bytewise	1.04	1.05	1.04	1.04	0.80	0.69	0.63	0.65	0.88	0.78	0.65	0.66
Mapped/Bytewise	1.07	1.10	1.08	1.07	0.69	0.63	0.59	0.61	0.75	0.65	0.61	0.61
Mapped/Charwise	1.02	1.04	1.03	1.03	0.87	0.92	0.95	0.94	0.85	0.84	0.94	0.93

(c) Construction time (ms)

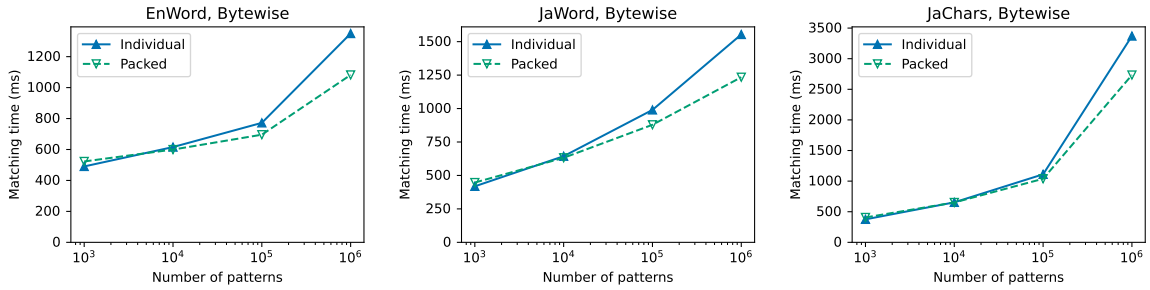
Scheme	EnWord				JaWord				JaChars			
	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Bytewise	0.12	1.34	21	244	0.13	1.44	27	387	0.09	1.2	20	275
Charwise	0.14	0.79	18	199	0.32	1.09	163	1,829	0.29	2.3	134	21,184
Mapped	0.09	0.81	18	204	0.06	1.50	102	1,251	0.08	1.4	45	9,107
Charwise/Bytewise	1.18	0.59	0.86	0.81	2.51	0.76	6.03	4.72	3.14	1.92	6.72	77.1
Mapped/Bytewise	0.80	0.60	0.86	0.83	0.44	1.04	3.77	3.23	0.83	1.22	2.25	33.1
Mapped/Charwise	0.68	1.03	1.01	1.03	0.17	1.37	0.62	0.68	0.26	0.64	0.33	0.43

JaChars, whose strings consist of multibyte characters, Charwise defines fewer states than Bytewise but more vacant ids. Especially in JaChars, the vacant proportion in Charwise is always more than 68%. The large vacant proportions are related to the average number of outgoing transitions from an internal state, as reported in Table 6c. The more outgoing transitions there are, the easier it is for vacant searches to fail and the number of vacant ids to increase. The large vacant proportions of Charwise, however, can be improved with the code mapping π, as the result of Mapped demonstrates.

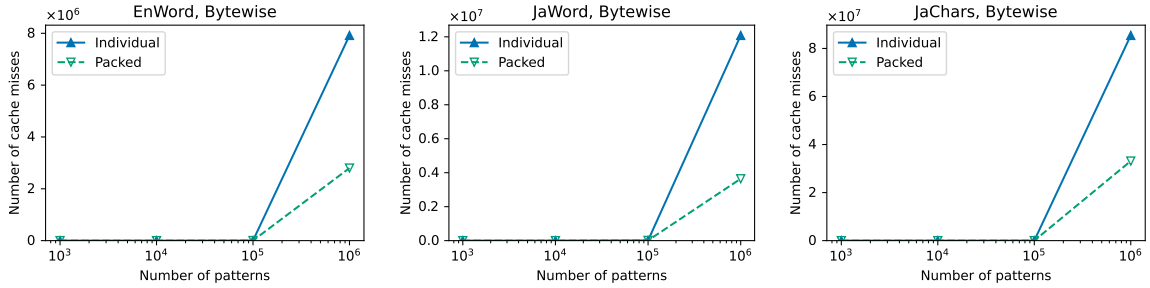
Table 6d reports the total memory usage of BASE, CHECK, FAIL, and OUTPOS (and the mapping π in Mapped). On EnWord, there is no significant difference between the schemes when the number of patterns is no less than 10K. On JaWord and JaChars, Mapped is always the first or second smallest because of its fewer states and smaller vacant proportions.

Table 7 shows the experimental results on time efficiency, where Table 7a reports the number of visiting states during matching with δ and f. On EnWord, whose characters are mostly single-byte ones, there is no significant difference between the schemes. On JaWord and JaChars, whose strings consist of multibyte characters, Charwise (or Mapped) achieves half as many as Bytewise, resulting in halving the number of random accesses during a pattern matching. Table 7b lists the elapsed time during a pattern matching. On JaWord and JaChars, Charwise and Mapped are faster than Bytewise because of their fewer random accesses. Comparing Charwise and Mapped, there is no large difference, although Mapped requires additional computations for the code mapping π.

Table 7c shows the elapsed time to compute two arrays, BASE and CHECK, from an original trie. On EnWord, Charwise and Mapped are faster than Bytewise in most cases. On JaWord and JaChars, Charwise is much slower than Bytewise, and its time performance is improved by the code mapping of Mapped; nevertheless, Mapped is still much slower than Bytewise for large dictionaries. The time inefficiency of Charwise and Mapped is caused by their large vacant proportion (as shown in Table 6b): Chain does not perform well when there are many vacant ids.



(a) Matching time (ms)



(b) Number of L3 cache misses occurring during a pattern matching

Figure 9: Comparison results on matching time for the Individual and Packed layouts.

4.4 Analysis on memory layouts

Figure 9 shows the experimental results for the Individual and Packed layouts presented in Section 3.3, while fixing the other settings to Forest, Bytewise, Basic, Chain, and LexDFS. Figure 9a shows the matching time. The greater the number of patterns, the faster Packed becomes. Packed is ≈20% faster than Individual when the number of patterns is 10^6. The time efficiency of Packed stems from its cache-efficiency. Figure 9b shows the number of L3 cache misses occurring during a pattern matching. When the number of patterns is no greater than 10^5, cache misses infrequently occur because the data structure fits in the cache memory. However, when the number of patterns is 10^6, Packed achieves 38–47% cache misses compared to Individual.

4.5 Analysis on array formats

Table 8 shows the experimental results for the Basic and Compact formats presented in Section 3.4, while fixing the other settings to Forest, Bytewise, Packed, Chain, and LexDFS.

Table 8a reports the total memory usage of BASE, CHECK, FAIL, and OUTPOS. In all cases, the memory usage in Compact is 75% of that in Basic because there is no difference in the resulting number of vacant ids (although Equation (5) has to be satisfied). There is no significant difference in the matching time, as Table 8b shows.

Table 8c reports the construction time. Compact is always slower because of Equation (5), especially on JaWord of 1M patterns. To clarify the slowdown, we investigated vacant ids for which vacant searches were mostly unsuccessful and found that the top-five vacant ids could only accept the five transition labels 0xC0, 0xCF, 0xD0, 0xD1, and 0xFF to satisfy Equation (5). Because these labels did not appear or were very few in the dictionary, vacant searches for the vacant ids were unsuccessful many times, resulting in the slow construction. Consequently, while the Compact format does not produce a particularly high number of vacant ids, some of them can significantly slow down vacant searches when using Chain.

Table 8: Experimental results for the Basic and Compact formats. The memory usage is the total of BASE, CHECK, FAIL, and OUTPOS. The last row in each table shows the different ratios for each result.

(a) Memory usage (KiB)

	EnWord				JaWord				JaChars			
Format	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Basic	44.0	404	3,712	33,276	48.0	440	4,780	76,544	32.0	292	2,908	27,972
Compact	33.0	303	2,784	24,957	36.0	330	3,585	57,408	24.0	219	2,181	20,979
Compact/Basic	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75

(b) Matching time (ms)

	EnWord				JaWord				JaChars			
Format	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Basic	523	600	696	1,082	447	632	879	1,234	406	651	1,038	2,733
Compact	545	606	712	1,052	473	633	893	1,226	415	609	1,028	2,439
Compact/Basic	1.04	1.01	1.02	0.97	1.06	1.00	1.02	0.99	1.02	0.94	0.99	0.89

(c) Construction time (ms)

	EnWord				JaWord				JaChars			
Format	1K	10K	100K	1M	1K	10K	100K	1M	1K	10K	100K	1M
Basic	0.11	1.06	19.2	207	0.13	1.19	22.4	331	0.08	0.81	14.8	200
Compact	0.14	1.30	23.5	249	0.14	1.39	32.7	182,015	0.09	0.91	16.0	226
Compact/Basic	1.25	1.23	1.22	1.20	1.09	1.17	1.46	549	1.09	1.12	1.08	1.13

Table 9: The average number of verifications per vacant search when using Chain. The number of patterns in each dictionary is one million.

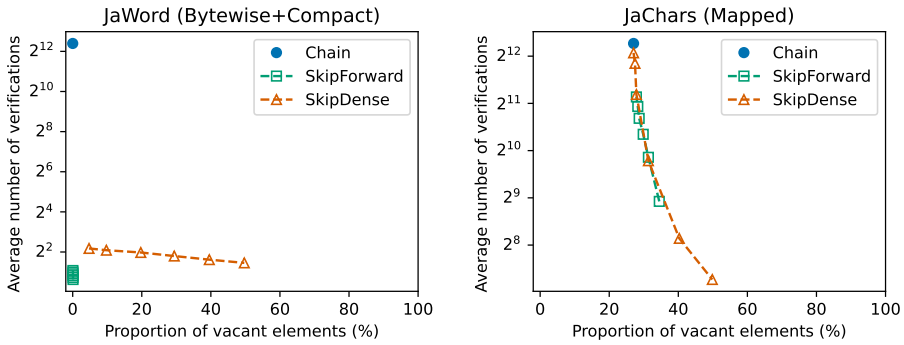
Method	EnWord	JaWord	JaChars
Bytewise + Basic	2.8	2.8	6.6
Bytewise + Compact	4.6	5383.3	7.4
Mapped	2.5	84.4	4934.5

4.6 Analysis on acceleration techniques for vacant searches

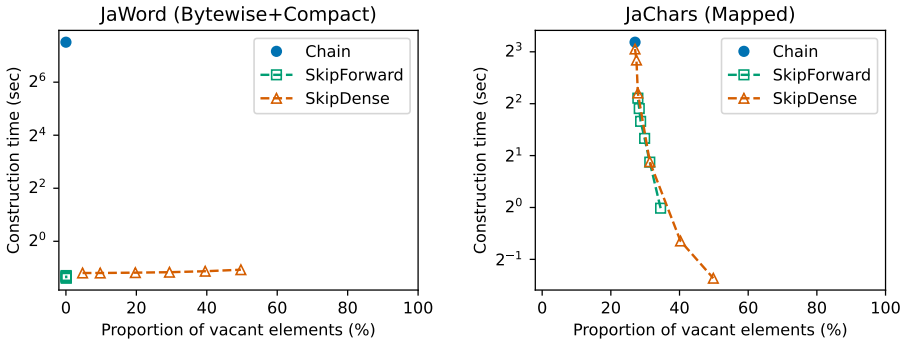
We first investigate problematic cases in DAAC construction when using Chain. Table 9 shows the average number of verifications per vacant search when using Chain. In the combination of Bytewise and Basic, the number is always several, indicating that it is hard to improve vacant searches further even using SkipForward or SkipDense. However, the number is significantly higher in the cases of (i) Compact on JaWord and (ii) Mapped on JaChars. Case (i) is because of Equation (5), as discussed in Section 4.5. Case (ii) is because of many vacant ids, as discussed in Section 4.3.

We next test SkipForward and SkipDense to improve the two cases. We test parameters $L = 4, 8, 12, 16, 20, 24$ in SkipForward and $\tau = 0.05, 0.1, 0.2, 0.3, 0.4, 0.5$ in SkipDense. Figure 10a shows the experimental results for the average number of verifications per vacant search. The left figure corresponds to case (i), where we can see that SkipForward achieves several verifications with any parameter L while maintaining similar vacant proportions. SkipDense also achieves several verifications, although the vacant proportions increase depending on the parameter τ. In the right figure corresponding to case (ii), both SkipForward and SkipDense achieve fewer verifications than Chain. SkipForward and SkipDense plot similar curves, and their performances have no significant difference.

Figure 10b shows the experimental results for the construction times. The plots are similar to those in Figure 10a, and both SkipForward and SkipDense achieve construction times within several seconds. These results indicate that we should choose SkipForward or SkipDense in accordance with the desired purpose: if we want to control construction times, SkipForward is best, and if we want to control memory usage, SkipDense is more suitable.



(a) The average number of verifications per vacant search



(b) Construction time in seconds

Figure 10: Experimental results for verifications and construction times when using Chain, SkipForward, or SkipDense.

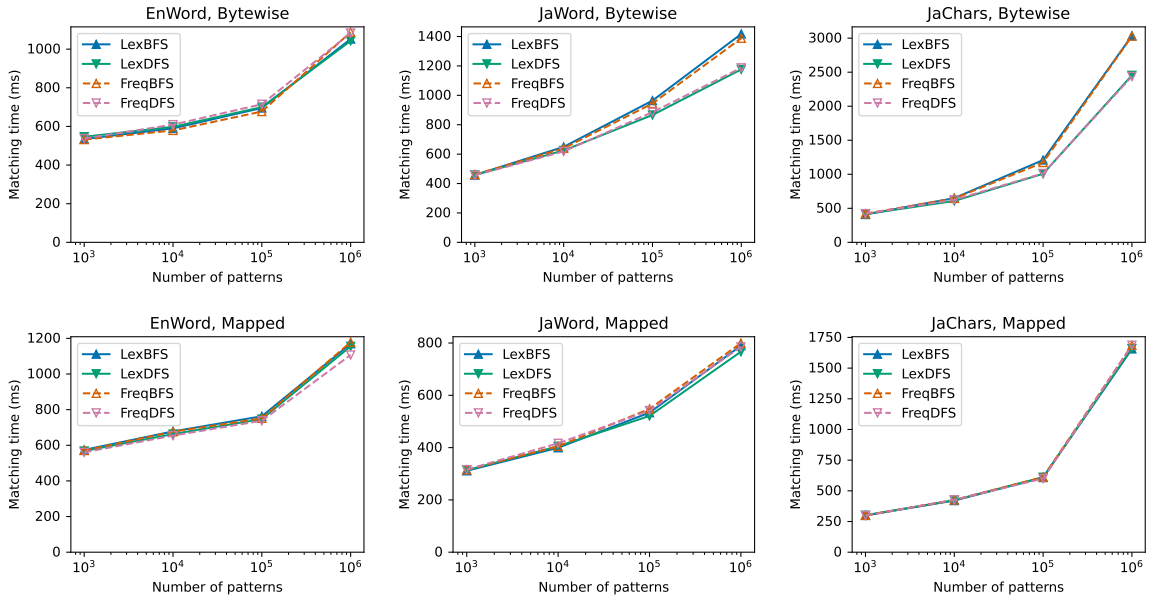


Figure 11: Matching times in milliseconds for different traversal orders.

4.7 Analysis on traversal orders

Figure 11 shows the experimental results of matching times for LexBFS, FreqBFS, LexDFS, and FreqDFS presented in Section 3.6. In the Bytewise scheme, we fix the other settings to Forest, Packed, Compact, and SkipForward ($L = 16$). In the Mapped scheme, we fix the other settings to Forest, Packed, and SkipForward ($L = 16$).

We compare the DFS and BFS orders. DFS is faster than BFS for JaWord and JaChars in Bytewise. To investigate the reason, we show the number of times states are visited in each depth during a pattern matching in Figure 12. As we can see, deeper states are often visited for JaWord and JaChars in Bytewise. The number of L3 cache misses occurring during a pattern matching is shown in Table 10, where we can see that DFS is cache-efficient in most cases. These observations indicate that DFS enables cache-efficient traversals on deeper states and is suitable for long patterns.

Comparing Lex and Freq, there is no significant difference. Although we also tested other orders, such as a random order, we did not observe any significant differences. The reason is explained by the average number of outgoing transitions for an internal state reported in Table 6c. The average number is the average size of E ($\neq \emptyset$) in Algorithm 2. In all cases, the average number is several, indicating that the order to visit elements in E is insignificant.

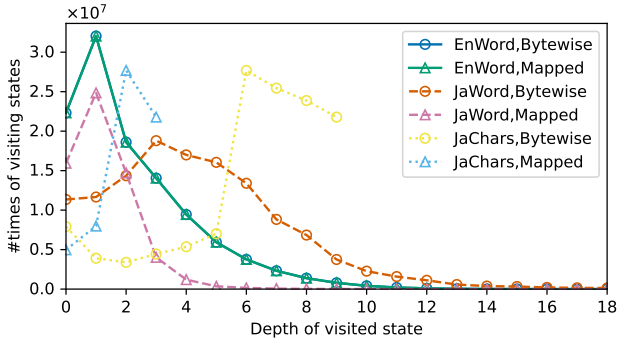


Figure 12: The number of visiting states at each depth during a pattern matching.

Table 10: The number of L3 cache misses occurring during a pattern matching with different traversal orders ($\times 10^3$). The number of patterns is fixed to 10^6. The last row shows the different ratios for each result.

Order	EnWord		JaWord		JaChars	
	Bytewise	Mapped	Bytewise	Mapped	Bytewise	Mapped
LexBFS	2,485	2,967	4,301	3,157	31,999	17,719
LexDFS	2,303	2,888	3,209	2,944	28,804	17,917
LexDFS/LexBFS	0.93	0.97	0.75	0.93	0.90	1.01

4.8 Comparison with other AC automata

On the basis of the above experimental analyses, we determine the best combinations of techniques and develop a new Rust library, called Daachorse, containing the following two variants:

- Bytewise-Daachorse = Forest + Bytewise + Packed + Compact + SkipForward ($L = 16$) + LexDFS
- Charwise-Daachorse = Forest + Mapped + Packed + SkipForward ($L = 16$) + LexDFS

We compare Daachorse with the *Aho-corasick* library [19], which is the most popular implementation of AC automata in Rust. This library provides two types of implementations: NFA-AC is a standard AC automaton (as described in Section 2.3), and DFA-AC is a deterministic finite automaton compiled from NFA-AC. DFA-AC is an option for faster matching, although it consumes a huge amount of memory.

Figure 13 shows the comparison results. For the matching times reported in Figure 13a, Bytewise-Daachorse is always the fastest on EnWord, and Charwise-Daachorse is always the fastest on JaWord and JaChars. Although DFA-AC is also fast when the number of patterns is 10^3, its performance degrades as the number of patterns increases. For the construction times reported in Figure 13b, NFA-AC is always the fastest, and Bytewise and Charwise-Daachorse approach NFA-AC. DFA-AC is always the slowest because compiling from NFA-AC takes time. For the memory usages reported in Figure 13c, Bytewise- and Charwise-Daachorse are often the smallest; however, when the number of patterns is 10^3, Charwise-Daachorse is larger than NFA-AC because of the memory consumption of the mapping π.

5 Application example: Japanese tokenization

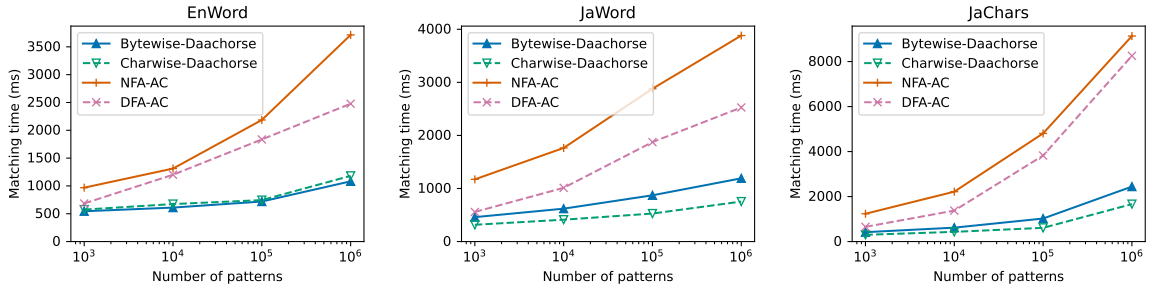
Vaporetto [10], a supervised Japanese tokenizer written in Rust, is an application example of AC automata. Vaporetto is an efficient implementation of the pointwise prediction algorithm [31] and leverages the AC algorithm in its feature extraction phase. To demonstrate the ability of Daachorse, we integrated the Daachorse and Aho-corasick libraries with Vaporetto and evaluated the tokenization speeds.

We conducted these experiments in the same environment as the experiments in Section 4. To train the Vaporetto model, we used 60K sentences with manually annotated short-unit-word boundaries in BCCWJ (version 1.1) [30] and 667K unique words in UniDic (version 3.1.0) [11], while specifying default parameters in Vaporetto. We performed tokenization for 5.9M sentences in BCCWJ that are not used for the training and then measured the running time.

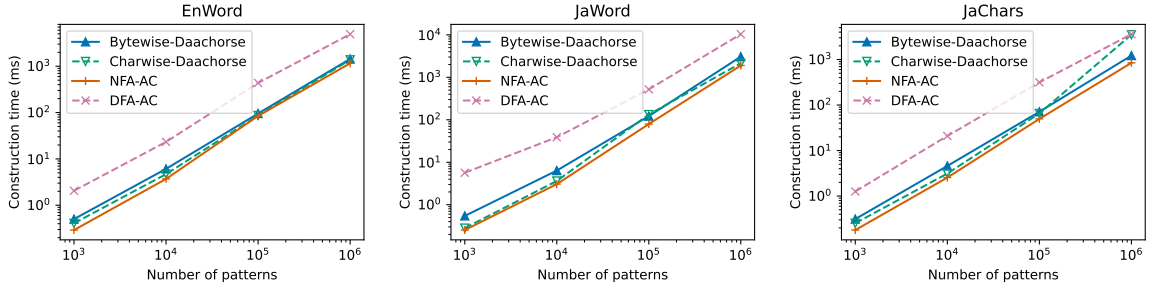
The experimental result showed that:

- Bytewise-Daachorse performed in 3.3 microseconds per sentence,
- Charwise-Daachorse performed in 2.9 microseconds per sentence,
- NFA-AC performed in 7.5 microseconds per sentence, and
- DFA-AC performed in 7.3 microseconds per sentence.

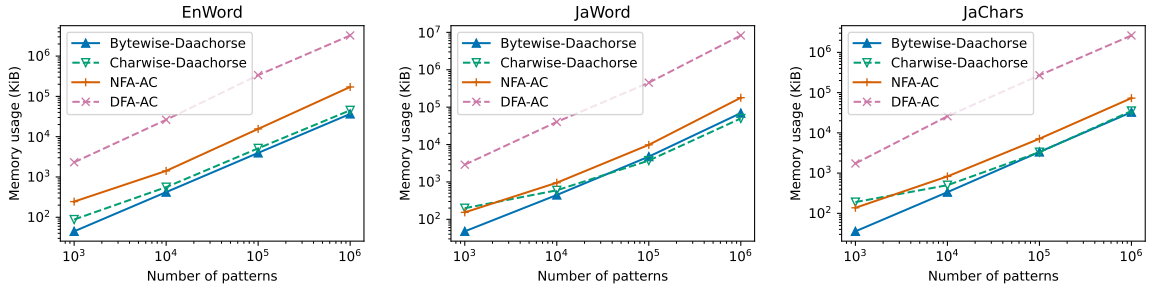
Daachorse was at most 2.6× faster than the Aho-corasick library. This result indicates that the time efficiency of the AC algorithm is critical in Vaporetto's tokenization, demonstrating the utility of our Daachorse.



(a) Matching time (ms)



(b) Construction time (ms)



(c) Memory usage (KiB)

Figure 13: Comparison results with the Daachorse and Aho-corasick libraries. (a) The matching time is plotted in a linear scale because the time complexity is not linear over the number of patterns. (b) The construction time and (c) the memory usage are plotted in a logarithmic scale because the complexities are linear over the number of patterns.

6 Related works

Double-array The two most popular implementations of DAACs are the Java library by hankcs [20] and the Go library by Ruohang [40]. Both libraries are implemented in a similar manner and employ the same techniques: Simple (although the memory layout is not identical), Charwise, Individual, SkipDense ($\tau = 0.05$, without Chain and the block partitioning), and LexBFS. As demonstrated in Section 4, these techniques do not provide the best performances in most cases, and the implementations could be improved.

Some studies have represented the integers of BASE and CHECK in a compressed space other than Compact (described in Section 3.4). Fuketa et al. [18] proposed partitioning one double-array structure into several smaller ones to implement BASE and CHECK consisting of 2-byte integers; however, as this approach produces many structures for a large trie, many pointers must be maintained to connect the structures. Fuketa et al. [17] proposed eliminating BASE by introducing additional character mappings, although its applications are limited to fixed-length keywords such as zip codes. Kanda et al. [25, 26] proposed approaches to compress the integers of BASE and CHECK through differential encoding; however, time efficiency can degrade because of more computations.

Another approach to compress the double-array is to employ compact trie forms. The *minimal-prefix (MP) trie* [3, 13] is an often-used form [2, 26, 49], and various methods to efficiently implement the MP-trie using the double-array have been proposed [12, 24, 48, 50]. However, the MP-trie is specialized for dictionary lookups, which are done by traversing a trie from the root to a leaf. Other compact forms such as double-tries [4] and directed acyclic word graphs [47] cannot be applied to AC automata in principle.

Liu et al. [29] showed that double-array structures resulting from large alphabets (e.g., multibyte characters) can include many vacant ids and are thus memory inefficient. They proposed several approaches to address this problem and demonstrated that code mapping with the frequency of characters is effective on Chinese strings. The efficiency of the code mapping was also demonstrated on N-gram sequences [36, 44].

Alternative trie representations Tries [16] have been studied since the 1960s, and there are many data structures to represent. Since the AC automaton is a simple extension of the trie, we have many alternatives to the double-array. The conventional data structures are a *matrix form* and a *list form* [6]. The matrix form uses a transition matrix of size $|S| \times |\Sigma|$ and performs a transition lookup in $O(1)$ time, while consuming a large space of $O(|S||\Sigma|)$. The list form stores a set of transition labels from each state in a sorted array and performs a transition lookup by binary search in $O(\log |\Sigma|)$ time, while consuming a smaller space of $O(|S|)$. The matrix and list forms have a time-space trade-off, and the double-array harnesses both advantages by compressing the matrix form.

There are other data structures that utilize both advantages of the matrix and list forms. One is a hybrid approach that uses the matrix form for states with many outgoing transitions and the list form for states with few outgoing transitions. The aho-corasick library [19] employs the hybrid approach and is outperformed by Daachorse, as we demonstrated. Another approach is hashing [6], which stores mappings from states to outgoing transitions in hash tables. This approach can perform a transition lookup in $O(1)$ expected time while consuming $O(|S|)$ space. However, searching in the hash table often requires more computations than transition lookups in the double-array. Prior experiments [35] demonstrated that DAACs outperformed AC automata with the hashing approach.

Compressed representations of tries have recently been proposed to store massive datasets in main memory. *Succinct tries* [5] are representative data structures. Their memory usage achieves $|S| \log_2 |\Sigma| + O(|S|)$ bits of space and is close to the information-theoretic lower bound [23]. However, the succinct tries employ many bit manipulations in tree navigational operations, and their time efficiency is not competitive with that of double-arrays [25, 26]. More compressed data structures, such as XBW [14] and Elias-Fano tries [38, 39], also have similar time bottlenecks and are not suited to design fast AC automata.

Compressed representations of AC automata Another line of research proposes data structures to represent AC automata (i.e., transition, failure, and output functions) in a compressed space. Belazzougui [7] proposed the first compressed data structure that supports a matching in optimal $O(n + occ)$ time, while achieving $|S| \log_2 |\Sigma| + O(|S|)$ bits of space. Hon et al. [21] achieved an entropy compressed space while matching time remains optimal. I et al. [22] designed a matching algorithm working on grammar-based compressed AC automata. However, these studies were accomplished through theoretical discussions, and we are unaware of any actual implementation.

7 Conclusion

In this paper, we provided a comprehensive description of implementation techniques in DAACs and experimentally revealed the most efficient combinations of the techniques to achieve higher performance. We also designed a data structure of DAACs and developed a new Rust library for faster multiple pattern matching, called Daachorse. Our experiments showed that, compared to other implementations of AC automata, Daachorse offered a superior performance in terms of time and space efficiency. As we demonstrated in the integration test with Vaporetto, Daachorse has significant potential to improve applications that employ multiple pattern matching. Our future work will incorporate Daachorse in other applications to enable faster text processing.

Acknowledgement

We thank Shinsuke Mori for his cooperation and Keisuke Goto for his insightful review.

References

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

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

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

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

[5] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane. Succinct trees in practice. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.

[6] Nikolas Askitis. *Efficient data structures for cache architectures*. PhD thesis, RMIT University, 2007.

[7] Djamal Belazzougui. Succinct dictionary matching with no slowdown. In *Proceedings of the 21st Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 88–100, 2010.

[8] Thorsten Brants and Alex Franz. Web 1T 5-gram Version 1. *Linguistic Data Consortium*, 2006.

[9] Brazil Inc. Groonga: An open-source fulltext search engine and column store (12.0.3). https://groonga.org/, 2022.

[10] daac-tools. Vaporetto: Very Accelerated POintwise pREdicTion based TOkenizer (0.4.0). https://github.com/daac-tools/vaporetto, 2022.

[11] Yasuharu Den, Toshinobu Ogiso, Hideki Ogura, Atsushi Yamada, Nobuaki Minematsu, Kiyotaka Uchimoto, and Hanae Koiso. The development of an electronic dictionary for morphological analysis and its application to Japanese corpus linguistics (in japanese). *Japanese linguistics*, 22:101–123, 2007.

[12] Tshering C Dorji, El-sayed Atlam, Susumu Yata, Mahmoud Rokaya, Masao Fuketa, Kazuhiro Morita, and Jun'ichi Aoe. New methods for compression of MP double array by compact management of suffixes. *Information Processing & Management*, 46(5):502–513, 2010.

[13] John A Dundas. Implementing dynamic minimal-prefix tries. *Software: Practice and Experience*, 21(10):1027–1040, 1991.

[14] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 2005.

[15] Paolo Ferragina and Gonzalo Navarro. Pizza&Chili Corpus. http://pizzachili.dcc.uchile.cl/texts.html, 2005.

[16] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[17] Masao Fuketa, Hiroya Kitagawa, Takuki Ogawa, Kazuhiro Morita, and Jun-ichi Aoe. Compression of double array structures for fixed length keywords. *Information processing & management*, 50(5):796–806, 2014.

[18] Masao Fuketa, Kazuhiro Morita, Toru Sumitomo, Shinkaku Kashiji, Elsayed Atlam, and Jun-Ichi Aoe. A new compression method of double array for compact dictionaries. *International Journal of Computer Mathematics*, 81(8):943–953, 2004.

[19] Andrew Gallant. aho-corasick: A fast implementation of Aho-Corasick in Rust (0.7.18). https://github.com/BurntSushi/aho-corasick, 2021.

[20] hankcs. AhoCorasickDoubleArrayTrie: An extremely fast implementation of Aho Corasick algorithm based on Double Array Trie (1.2.2). https://github.com/hankcs/AhoCorasickDoubleArrayTrie, 2020.

[21] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, 475:113–119, 2013.

[22] Tomohiro I, Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Compressed automata for dictionary matching. *Theoretical Computer Science*, 578:30–41, 2015.

[23] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[24] Shunsuke Kanda, Yuma Fujita, Kazuhiro Morita, and Masao Fuketa. Practical rearrangement methods for dynamic double-array dictionaries. *Software: Practice and Experience*, 48(1):65–83, 2018.

[25] Shunsuke Kanda, Masao Fuketa, Kazuhiro Morita, and Jun'ichi Aoe. A compression method of double-array structures using linear functions. *Knowledge and Information Systems*, 48(1):55–80, 2016.

[26] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems*, 51(3):1023–1042, 2017.

[27] Taku Kudo, Kaoru Yamamoto, and Yuji Matsumoto. Applying conditional random fields to Japanese morphological analysis. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 230–237, 2004.

[28] Takuu Kudo. Darts: Double-ARray Trie System (0.32). http://chasen.org/~taku/software/darts/, 2008.

[29] Huidan Liu, Minghua Nuo, Long-Long Ma, Jian Wu, and Yeping He. Compression methods by code mapping and code dividing for chinese dictionary stored in a double-array trie. In *Proceedings of the 5th International Joint Conference on Natural Language Processing (IJCNLP)*, pages 1189–1197, 2011.

[30] Kikuo Maekawa, Makoto Yamazaki, Toshinobu Ogiso, Takehiko Maruyama, Hideki Ogura, Wakako Kashino, Hanae Koiso, Masaya Yamaguchi, Makiro Tanaka, and Yasuharu Den. Balanced corpus of contemporary written Japanese. *Language resources and evaluation*, 48(2):345–371, jun 2014.

[31] Shinsuke Mori, Yosuke Nakata, Graham Neubig, and Tatsuya Kawahara. Morphological Analysis with Pointwise Predictors (in Japanese). *Journal of Natural Language Processing*, 18(4):367–381, 2011.

[32] Kazuhiro Morita, Masao Fuketa, Yoshihiro Yamakawa, and Jun'ichi Aoe. Fast insertion methods of a double-array structure. *Software: Practice and Experience*, 31(1):43–65, 2001.

[33] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[34] Graham Neubig, Yosuke Nakata, and Shinsuke Mori. Pointwise prediction for robust, adaptable Japanese morphological analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 529–533, 2011.

[35] Janne Nieminen and Pekka Kilpeläinen. Efficient implementation of Aho–Corasick pattern matching automata using unicode. *Software: Practice and Experience*, 37(6):669–690, 2007.

[36] Jun'ya Norimatsu, Makoto Yasuhara, Toru Tanaka, and Mikio Yamamoto. A fast and compact language model implementation using double-array structures. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 15(4):27, 2016.

[37] Masaki Oono, El-Sayed Atlam, Masao Fuketa, Kazuhiro Morita, and Jun'ichi Aoe. A fast and compact elimination method of empty elements from a double-array structure. *Software: Practice and Experience*, 33(13):1229–1249, 2003.

[38] Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 615–624, 2017.

[39] Giulio Ermanno Pibiri and Rossano Venturini. Handling massive N-gram datasets efficiently. *ACM Transactions on Information Systems*, 37(2), feb 2019.

[40] Feng Ruohang. ac: Aho-Corasick Automaton with Double Array Trie. https://github.com/Vonng/ac, 2019. The latest version at May 24, 2022.

[41] Manabu Sassano. An empirical study of active learning with support vector machines for Japanese word segmentation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 505–512, 2002.

[42] Hiroyuki Shinnou. Deterministic japanese word segmentation by decision list method. In *Proceedings of the 6th Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, pages 822–822, 2000.

[43] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast wordpiece tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2089–2103, 2021.

[44] Makoto Yasuhara, Toru Tanaka, Jun ya Norimatsu, and Mikio Yamamoto. An efficient language model using double-array structures. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 222–232, 2013.

[45] Susumu Yata. Nihongo Web Corpus 2010 (NWC 2010). http://www.s-yata.jp/corpus/nwc2010/, 2010.

[46] Susumu Yata. A clone of Darts. https://github.com/s-yata/darts-clone, 2018. The latest version at May 24, 2022.

[47] Susumu Yata, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. Fast string matching with space-efficient word graphs. In *Proceedings of the 4th International Conference on Innovations in Information Technology (IIT)*, pages 79–83, 2008.

[48] Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, and Jun'ichi Aoe. An efficient deletion method for a minimal prefix double array. *Software: Practice and Experience*, 37(5):523–534, 2007.

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

[50] Susumu Yata, Masaki Oono, Kazuhiro Morita, Toru Sumitomo, and Jun'ichi Aoe. Double-array compression by pruning twin leaves and unifying common suffixes. In *Proceedings of the 1st International Conference on Computing & Informatics (ICOCI)*, pages 1–4, 2006.

[51] Zhou Yihan. Meaningfulness and unit of Zipf's law: evidence from danmu comments. In *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, pages 1046–1057, August 2021.

[52] Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 1091–1102, 2014.