Vol.2018-DBS-167 No.15
Vol.2018-IFAT-132 No.15
2018/9/13

ダブル配列オートマトンによる圧縮文字列辞書の実装

松本 拓真[1,a)]　神田 峻介[2,b)]　森田 和宏[1,c)]　泓田 正雄[1,d)]

概要：文字列辞書は，文字列集合を保管し検索・復元機能を備えるデータ構造であり，近年，様々な用途でそのコンパクト性が求められている．辞書を実現するための優れた技法として Trie などがあり，これらを効率よく表現するデータ構造が多く提案されている．本稿では，文字列集合の接頭辞と接尾辞を効率的に圧縮できる DFA(決定性有限オートマトン) を用いた圧縮文字列辞書を提案する．DFA の表現に用いるデータ構造にはダブル配列オートマトンを採用し，辞書の機能を実現するための実装と，それに伴う圧縮手法を紹介する．提案手法は文字列の復元時間に理論的課題を有していたものの，実データを用いた実験では，メモリ効率と検索性能のトレードオフにおいて他の手法と同等の性能を示しつつ，高い検索性能を持つことを示した．

キーワード：圧縮文字列辞書，決定性有限オートマトン，ダブル配列

1. はじめに

文字列辞書は，文字列の集合と，その各文字列に対する一意の ID を保管するデータ構造である．即ち，文字列の入力に対しその ID を報告する Lookup と，ID の入力に対しその文字列を報告する Access の二つの操作を提供するデータ構造である．文字列辞書は，自然言語処理や情報検索などの数多くの用途で用いられている．一方で，近年では大規模なデータに対する辞書サイズの大きさが問題視されており [1]，辞書の圧縮表現である圧縮文字列辞書の提案が多くなされている [1–4].

圧縮文字列辞書を実現する技法の一つに Trie がある．Trie は，文字列集合の接頭辞を効率的に圧縮し，圧縮文字列辞書を実現できる．Trie を表現するデータ構造には，圧縮率に優れた LOUDS や，検索速度に優れたダブル配列などがある．一方で，DFA(決定性有限オートマトン) [5] は，文字列集合の接頭辞と接尾辞を効率的に圧縮できるが，圧縮文字列辞書への適応事例はない．

本稿では，圧縮文字列辞書を DFA により実現するデータ構造を提案する．ベースとなるデータ構造としてダブル配列オートマトン [6] を採用し，Lookup/Access を実行するための実装と，それに伴った効率的な圧縮手法を説明する．最後に提案手法の有効性を実データを用いた実験により示し，他の手法との比較により評価を与える．

1 徳島大学大学院先端技術科学教育部
2 理化学研究所 革新知能統合研究センター
a) tkm.matsumoto@gmail.com
b) shunsuke.kanda@riken.jp
c) kam@is.tokushima-u.ac.jp
d) fuketa@is.tokushima-u.ac.jp

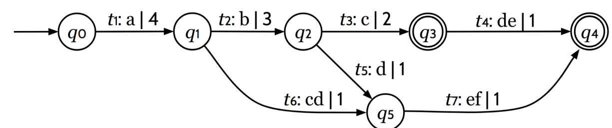

図 1　キー集合 K を表す DFA

2. DFA による辞書の実現

文字列辞書を DFA で実現する際に，文字列によりラベル付けされた遷移を辿り，初期状態から受理状態までの経路により文字列を表すことにする．また，各遷移に対し，その遷移以降に現れる経路の数を保存する．即ちこの値は，遷移後に現れる可能性のある文字列のパターン数を表していることになる．以後この値を，遷移のワード数と表記する．遷移にワード数を示すことにより，DFA 上に現れる全ての経路を一意の番号で表すことができるようになる [7]．つまり，DFA 内に保存される文字列に対し一意の ID を割り当てることができる．例として，キー集合 $K = \{$“abc”, “abcde”, “abdef”, “acdef”$\}$ を表す DFA を図 1 に示す．図において，初期状態は q_0 である．二重丸で表した状態は受理状態であり，その状態に至る全ての経路で表される文字列が辞書中に存在することを

ⓒ 2018 Information Processing Society of Japan

表している．また，遷移 t_2 以降に現れる経路パターンは $\{t_3\}, \{t_3, t_4\}, \{t_5, t_7\}$ の 3 つであるため，遷移 t_2 のワード数は 3 となる．

2.1 DFA における Lookup

文字列集合は辞書順昇順に並んでいるとし，i 番目の文字列を $word_i$ で表す．操作を行う前に，カウンター C を初期値 1 で準備する．$word_i$ を入力文字列とし，$word_i$ に従って DFA 上を遷移していき，各状態が持つ遷移を遷移ラベルの辞書順で確認する．経路上の各状態での操作を以下に示す．

- 次の経路となる遷移に到達するまで遷移をスキップし，スキップした全ての遷移のワード数を C に加算する
- 受理状態を通過した場合，C を 1 加算する

$word_i$ を受理するまで遷移を繰り返すことで，C は $word_i$ の文字列集合中の辞書順の順番を表すようになり，$i = C$ を満たしている．よって操作後の C を報告することで，$word_i$ から i が報告される．

2.2 DFA における Access

IDi から，文字列集合中の辞書順 i 番目の文字列 $word_i$ を復元する操作を以下に示す．ここで，遷移 t_j に保存される数値を V_j で表す．

- カウンター C の初期値を i とする
- 初期状態からそれぞれの状態において，遷移ラベルの辞書順で遷移を確認し，遷移の数値 V_j と C を比較し以下の操作を行う
  - $V_j < C$ の場合，t_j はスキップされ，C から V_j を減算する
  - $V_j \geq C$ の場合，t_j により次の状態へ遷移する．受理状態に到達した場合，C を 1 減算する．ここで $C = 0$ となった場合，現在の状態までにたどった経路上のラベルを復元し，報告する

3. 準備

本設では，辞書を実装する上で必要になるデータ構造を簡単に紹介する．はじめに，基本的な定義を以下に与える．文字がとり得る値の集合 Σ をアルファベットといい，そのサイズを $\sigma = |\Sigma|$ で表す．対数の底は 2 で統一する．

Rank 辞書　ビット列 $B[1, n]$ に対し，Rank と呼ばれる操作を実現する．Rank は，$o(n)$ ビットの補助データ構造を追加することにより，定数時間で実行できる [8].

- $\mathrm{Rank}(B, i)$: $B[1, i)$ 中の 1 の数を返す

4. 辞書 DFA のダブル配列オートマトン表現

ダブル配列オートマトンは，二つの一次元配列 NEXT と CHECK を用いて DFA を表現できる，検索速度に秀でたデータ構造である．NEXT と CHECK の各要素は DFA の遷移と対応しており，NEXT は遷移先の状態番号，CHECK は遷移ラベルの文字を保存している．受理状態の表現には，各要素に 1 ビットの記憶量を割り当てることで，受理状態に向かう遷移を区別する．

基本的なダブル配列の CHECK は一文字分の記憶量しか有していないため，遷移ラベルを文字列で表現することができない．そこで，遷移文字列を整数値に符号化する辞書符号化 [2,9] を行い，符号を CHECK に保存することで文字列による遷移を表現する手法が提案されている [10]．この手法をベースに圧縮文字列辞書を実装する．

まずベースとなるデータ構造を紹介し，次に圧縮文字列辞書を実現するための実装を説明し，構築されたデータ構造をまとめる．また，実装に伴う圧縮手法を説明する．

4.1 辞書符号化を用いたダブル配列オートマトン

辞書符号化のため，配列 STR を導入し，遷移文字列を連続して格納する．連続する文字列を区別するため，文字列の終端に終端文字 ‘#’ を格納する．そして，それぞれの文字列の先頭位置の添字を文字列の符号とし，この符号を遷移に対応する CHECK に保存する．遷移ラベルが一文字で構成される場合は符号化は行わず，そのまま CHECK に保存する．即ち，CHECK には文字と文字列の符号がそれぞれ保存されることになるため，ビット列 ID を導入し文字と符号を区別する．ダブル配列における状態間の遷移を式で示す．状態 u から状態 v への文字 c による遷移を式 (1) に示し，文字列 $s[1, l]$ による遷移を式 (2) に示す．

$$\begin{cases} e \leftarrow u + c \\ \mathrm{ID}[e] = 0 \\ \mathrm{CHECK}[e] = c \\ v \leftarrow \mathrm{NEXT}[e] \end{cases} \tag{1}$$

$$\begin{cases} e \leftarrow u + s[1] \\ \mathrm{ID}[e] = 1 \\ i \leftarrow \mathrm{CHECK}[e] \\ \mathrm{STR}[i, i + l - 1) = s[1, l) \\ \mathrm{STR}[i + l] = \text{‘\#’} \\ v \leftarrow \mathrm{NEXT}[e] \end{cases} \tag{2}$$

例として，キー集合 $K = \{$“abc”, “abcde”, “abdef”, “acdef”$\}$ に対する辞書符号化を用いたダブル配列オートマトン表現を図 2 に示す．図において，$\Sigma = \{$‘#’, ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’$\}$ は $\{0, 1, 2, 3, 4, 5, 6\}$ で表される．NEXT の要素のうち受理状態を示す状態番号は負の値で表現している．ダブル配列の先頭の要素は，初期状態への遷移を表している．

記憶量について，NEXT の値はダブル配列のいずれかの要素の添字を示すため，ダブル配列の要素数 n に依存し各要素 $\lceil \log n \rceil$ ビットである．CHECK は文字と文字列の符号を格納するため，双方を保存できるように記憶量を設定した場合，各要素 $max(\lceil \log \sigma \rceil, \lceil \log |\mathrm{STR}| \rceil)$ ビットを要

ⓒ 2018 Information Processing Society of Japan

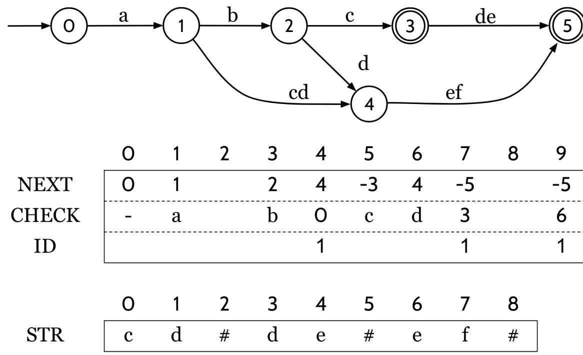

図 2 キー集合 K に対する
辞書符号化を用いたダブル配列オートマトン表現

することになる．しかし基本的に $\sigma \ll |\mathrm{STR}|$ であり，文字を保存する要素では $\lceil \log |\mathrm{STR}| \rceil - \lceil \log \sigma \rceil$ ビットの無駄な領域が存在することになる．そこで，CHECK の記憶量を各要素 $\lceil \log \sigma \rceil$ ビットとし，符号の上位 $\lceil \log \sigma \rceil$ はそのまま CHECK に保存する．配列 ID-FLOW を導入し，符号の下位 $\lceil \log |\mathrm{STR}| \rceil - \lceil \log \sigma \rceil$ ビットを連続して格納することで圧縮を行う．ダブル配列の要素 e の符号 i_e の復元は，ビット列 ID に対して Rank 辞書を構築し，式 (3) により定数時間で実行できる．よって辞書符号化を用いたダブル配列オートマトンの記憶量は，$u = \mathrm{Rank}(\mathrm{ID}, n)$，STR の要素数を l とすると，全体で $n(\lceil \log n \rceil + \lceil \log \sigma \rceil + 1) + u(\lceil \log l \rceil - \lceil \log \sigma \rceil) + l(\lceil \log \sigma \rceil) + o(n)$ ビットになる．

$$i_e \leftarrow \mathrm{CHECK}[e] + 2^{\lceil \log \sigma \rceil} \mathrm{ID\text{-}FLOW}[\mathrm{Rank}(\mathrm{ID}, e)] \quad (3)$$

4.2 Lookup の実装

DFA の各経路を一意の番号で表現するため，ダブル配列に配列 WORDS を導入し，各遷移に節 2 で述べたワード数を保存する．ダブル配列では連続した要素が同じ状態からの遷移であるとは限らないため，任意の状態からの遷移となる要素を走査する場合，次の経路上の遷移ラベルを $s[1, l]$ とすると，文字 c $(c \in \Sigma, c < s[1])$ による遷移の全ての要素において，遷移文字が正しいかどうかを確認してから WORDS を加算する必要がある．これは，各遷移でのワード数の加算には，最大 $\sigma - 1$ 回の計算が必要であることを意味する．よって WORDS を用いた Lookup の時間計算量は，入力文字列長 m とアルファベットサイズ σ に依存し，$O(m\sigma)$ となる．記憶量について，WORDS の取り得る値は辞書に含まれる文字列の数 q 以下であるため，WORDS の各要素は $\lceil \log q \rceil$ ビットの記憶量を要する．WORDS を用いた Lookup アルゴリズムを Algorithm1 に示す．入力文字が DFA に存在しない場合は，検索の失敗を報告する．また，遷移 e が受理状態への遷移であるかどうかを区別するビット列を ACCEPT で表現している．

クエリ検索における $O(m\sigma)$ という時間計算量は非常に遅いため，これを改善する．入力文字列による検索経路は一意に定まっているため，経路上の次の遷移のラベルを $s[1, l]$ とし，各状態が持つ遷移のラベルの先頭文字の集合 B $(B \subseteq \Sigma)$ の内，文字 b $(b \in B, b < s[1])$ による遷移のワード数の累積和を予め計算した値を保存しておくことで，定数時間で遷移が可能になる．そこで，各遷移の累積ワード数を保存する C-WORDS を導入する．累積ワード数を用いた Lookup の時間計算量は，入力文字列長 m のみに依存し，$O(m)$ となる．記憶量についても，各状態が持

Algorithm 1 ダブル配列オートマトンの WORDS を用いた Lookup アルゴリズム

```
 1: function TRANSLATE(e, query[0, m − 1], k)
 2:     e ← NEXT[e] + query[k]
 3:     if ID[e] = 0 then
 4:         if CHECK[e] ≠ query[k] then return false
 5:         return (e, k + 1)
 6:     else
 7:         i ← CHECK[e]
 8:         l ← 0
 9:         while STR[i + l] ≠ '#' do
10:             if STR[i + l] ≠ query[k + l] then return false
11:             l ← l + 1
12:         return (e, k + l)
13:
14: function LOOKUP(query[0, m − 1])
15:     e ← 0
16:     k ← 0
17:     c ← 1
18:     while k < m do
19:         if ACCEPT[e] = 1 then c ← c + 1
20:         for all q ∈ Σ (q < query[k]) do
21:             ce ← NEXT[e] + q
22:             if (ID[ce] = 0 and CHECK[ce] ≠ q) or
                    (ID[ce] = 1 and STR[CHECK[ce]] ≠ q) then
23:                 continue
24:             c ← c + WORDS[ce]
25:         trans ← TRANSLATE(e, query, k)
26:         if trans = false then return false
27:         (e, k) ← trans
28:     if ACCEPT[e] = 1 then return c
29:     else return false
```

Algorithm 2 ダブル配列オートマトンの C-WORDS を用いた Lookup アルゴリズム

```
 1: function LOOKUP_FAST(query[0, m − 1])
 2:     e ← 0
 3:     k ← 0
 4:     c ← 1
 5:     while k < m do
 6:         if ACCEPT[e] = 1 then c ← c + 1
 7:         trans ← TRANSLATE(e, query, k)
 8:         if trans = false then return false
 9:         (e, k) ← trans
10:         c ← c + C-WORDS[e]
11:     if ACCEPT[e] = 1 then return c
12:     else return false
```

つ遷移のワード数の合計が辞書の文字列数 q を超えることはないため，WORDS と同様に，C-WORDS の各要素は $\lceil \log q \rceil$ ビットを要する．C-WORDS を用いた Lookup アルゴリズムを Algorithm2 に示す．

4.3 Access の実装

DFA での Access には，入力 ID をもとに DFA 上を遷移し，経路上のラベルを復元することで ID から文字列を報告する．DFA の各状態では，遷移のワード数とカウンター C を比較しながら，遷移先を決定する．よって，節 4.2 で導入した WORDS を用いてダブル配列オートマトンにおける Access を実現する．しかしダブル配列では，節 4.2 で述べた理由により，各状態からの全ての遷移の探索には最大 σ 回の計算が必要となる．よって，WORDS を用いた Access の時間計算量は $O(m\sigma)$ である．WORDS を用いた Access アルゴリズムを Algorithm3 に示す．

$O(m\sigma)$ という時間計算量は非常に遅いため，これを改善する．アルファベット Σ を逐次的に探索して遷移先を決定する場合，遷移先として存在しない文字での遷移の確認の計算が無駄になっているため，任意の状態からの遷移ラベルの集合 B $(B \subseteq \Sigma)$ に対して逐次的に探索するのが望ましい．そこで，各状態における辞書順で最も小さい遷移ラベルの文字を各状態に保存し，また各状態の任意の遷移より辞書順で次に現れる遷移ラベルを各遷移に保存することで，任意の状態からの遷移を連結する [11]．各状態から最初に現れる遷移文字を保存する配列 ELD と，各遷移の次に現れる遷移文字を保存する配列 BRO を導入する．遷移ラベルが文字列であった場合は，先頭の文字を保存する．これらにより，各状態での遷移先の決定は最大でも $b = |B|$ 回の計算で実行できるようになり，WORDS,ELD,BRO を用いた Access の時間計算量は $O(mb)$ になる．記憶量について，各状態からの遷移の連結を文字によって表現するため，全ての遷移に対し $\lceil \log \sigma \rceil$ ビットの記憶量を追加することになる．しかし，ELD と BRO をそのまま導入した場合，各要素 $2\lceil \log \sigma \rceil$ ビットの記憶量が追加されることになるため，ELD と BRO の合計の約半分の記憶量が無駄になる．そこで，ELD と BRO の要素は空き要素を詰めて配置し，ダブル配列の各要素に ELD と BRO の存在の可否を表すビット列 fELD と fBRO を導入する．fELD と fBRO それぞれに Rank 辞書を構築することで，ELD の値 eld_e と BRO の値 bro_e はそれぞれ以下の式で定数時間で復元することができる．

$$eld_e \leftarrow \mathrm{ELD}[\mathrm{Rank}(\mathrm{fELD}, e)] \ (\mathrm{fELD}[e] = 1) \quad (4)$$

$$bro_e \leftarrow \mathrm{BRO}[\mathrm{Rank}(\mathrm{fBRO}, e)] \ (\mathrm{fBRO}[e] = 1) \quad (5)$$

よって，ELD と BRO は全体で $n(2 + \lceil \log \sigma \rceil) + o(n)$ ビットの記憶量で表現できる．WORDS,ELD,BRO を用いた Access アルゴリズムを Algorithm4 に示す．

Algorithm 3 ダブル配列オートマトンの WORDS を用いた Access アルゴリズム

```
 1: function Label(e)
 2:     if ID[e] = 0 then
 3:         return CHECK[e]
 4:     else
 5:         label ← “”
 6:         i ← CHECK[e]
 7:         while STR[i] ≠ ‘#’ do
 8:             label ← label + STR[i]
 9:             i ← i + 1
10:         return label
11:
12: function Access(id)
13:     e ← 0
14:     key ← “”
15:     c ← id
16:     while c > 0 do
17:         for all q ∈ Σ do
18:             ce ← NEXT[e] + q
19:             if (ID[ce] = 0 and CHECK[ce] ≠ q) or
                    (ID[ce] = 1 and STR[CHECK[ce]] ≠ q) then
20:                 continue
21:             if WORDS[ce] > c then
22:                 c ← c − WORDS[ce]
23:             else
24:                 e ← ce
25:                 break
26:         if ACCEPT[e] = 1 then
27:             c ← c − 1
28:         key ← key + Label(e)
29:     return key
```

Algorithm 4 ダブル配列オートマトンの WORDS,ELD,BRO を用いた Access アルゴリズム

```
 1: function Access_Fast(id)
 2:     e ← 0
 3:     key ← “”
 4:     c ← id
 5:     while c > 0 do
 6:         s ← NEXT[e]
 7:         ce ← s + ELD[s]
 8:         while WORDS[ce] > c do
 9:             c ← c − WORDS[ce]
10:             ce ← s + BRO[ce]
11:         e ← ce
12:         if ACCEPT[e] = 1 then
13:             c ← c − 1
14:         key ← key + Label(e)
15:     return key
```

4.4 構築されるデータ構造

本節で紹介した Lookup,Access の実装により構築されるダブル配列オートマトンをまとめる．キー集合 $K = \{$“abc”, “abcde”, “abdef”, “acdef”$\}$ に対する圧縮文字列辞書のダブル配列オートマトン表現を図 3 に示す．図の DFA の遷移の表示は，{ 遷移ラベル | ワード数 | 累積ワード数 }

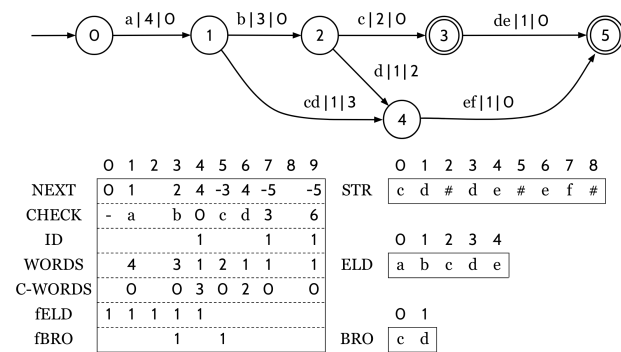

図 3 キー集合 K に対する
圧縮文字列辞書のダブル配列オートマトン表現

に対応している．ID,fELD,fBRO に対してはそれぞれ Rank 辞書を構築している．

4.5 WORDS,C-WORDS の圧縮

Access で用いる WORDS と，Lookup で用いる C-WORDS の各要素の記憶量は，辞書の文字列数 q に依存し $\lceil \log q \rceil$ ビットである．これらをそのまま保存した場合，$n(2\lceil \log q \rceil)$ ビットと無視できない記憶量を追加することになり，圧縮文字列辞書のコンパクト性が失われる，そこで，WORDS の各要素の記憶量は 4 ビットとし，ワード数 w の上位 4 ビットを WORDS に保存する．ここで配列 WORDS-FLOW を導入し，$w \geq 2^4$ となる w の下位 $\lceil \log q \rceil - 4$ ビットを連続して保存することで圧縮する．また，$w \geq 2^4$ であるかどうかを区別するビット列 fWORDS を導入する．WORDS の値 w_e の復元は，fWORDS に Rank 辞書を構築することで，以下の式で定数時間で実行できる．

$$\begin{cases} \text{fWORDS}[e] = 0 \\ w_e \leftarrow \text{WORDS}[e] \end{cases} \tag{6}$$

$$\begin{cases} \text{fWORDS}[e] = 1 \\ w_e \leftarrow \text{WORDS}[e] \\ \qquad +2^4\text{WORDS-FLOW}[\text{Rank}(\text{fWORDS}, e)] \end{cases} \tag{7}$$

C-WORDS に対しても，WORDS と同様に C-WORDS-FLOW と fC-WORDS を導入して圧縮する．元の WORDS 及び C-WORDS の各要素が 2^4 より小さい値であれば，それぞれ $\lceil \log q \rceil - 4$ ビット削減して表現できる．追加されるビット列 fWORDS,fC-WORDS の記憶量に対し，削減される記憶量が全体として多くなれば圧縮が実現できる．DFA の各状態において，親の遷移のワード数は子の遷移のワード数の合計になるため，DFA の後半に現れる遷移のワード数はほとんど小さい値で表現される．そのため，圧縮表現により WORDS と C-WORDS の多くの要素での圧縮が見込まれる．それぞれ上位 4 ビットで分割した理由は，WORDS と C-WORDS の上位 4 ビットを合わせて 8 ビットで表現でき，計算機において効率的だからである．

表 1 実験に用いたコーパスの詳細

	サイズ [MB]	文字列数 [x1,000]	平均長 [bytes/key]
C_1	日本語 Wikipedia の見出し集合 (2015 年 1 月時点)*1		
	32.3	1,518	22.31
C_2	インドシナ諸島のドメイン上でクロールし得られた URL 集合*2		
	612.9	7,415	86.68
C_3	日本語ウェブコーパス 2010 の N-gram コーパス*3		
	460.8	20,723	21.23

5. 評価

辞書 DFA のダブル配列オートマトン表現を C++により実装し，表 1 に示すコーパスから構築した辞書に対し，コーパスサイズに対する圧縮率と Lookup/Access 時間について他のデータ構造と比較する．実験に用いた計算機の構成は，Intel Core i7 4 GHz CPU，16 GB RAM であり，OS は macOS High Sierra 10.13.6 である．

本節では，提案手法を DAA と表記し，{C,L,i,w} からなるオプションにより提案手法の実装を区別する．オプションの詳細を以下に示す．

- -(none): WORDS,C-WORDS,ELD,BRO を実装，
  Lookup を C-WORD で実行，
  Access を WORDS,ELD,BRO で実行，
  WORDS,C-WORDS に圧縮を適応
- -C: C-WORDS を非実装，Lookup を WORDS で実行
- -L: ELD,BRO を非実装，Access を WORDS で実行
- -w: WORDS,C-WORDS を非圧縮

また比較手法として，Xor 圧縮を用いたダブル配列トライ [4] (XCDA *4)，Centroid Path-Decomposed Trie の Re-Pair を用いた圧縮表現 [3] (CentRP*5)，Prefix/Patricia Trie の入れ子による辞書の LOUDS 表現 [2] (Marisa*6) のデータ構造でも同様に実験を行い，圧縮率と Lookup/Access 時間を計測する．実験結果を表 2 に示す．提案手法と比較手法それぞれの結果のうち最も良い結果を太字で表記している．また，提案手法の結果のうち，最も悪い結果に下線を表記している．

記憶量について，WORDS 及び C-WORDS の圧縮については，C_1 では約 0.7 倍，C_2 では約 0.8 倍，C_3 では約 0.6 倍に圧縮できており，有効性が確認できる．圧縮済みの提案手法 DAA を他のデータ構造と比較した場合，C_1 ではい

*1 https://dumps.wikimedia.org/jawiki/
*2 http://data.law.di.unimi.it/webdata/indochina-2004/indochina-2004.urls.gz
*3 http://s-yata.jp/corpus/nwc2010/ngrams/
*4 https://github.com/kampersanda/xcdat
*5 https://github.com/ot/path_decomposed_tries
*6 https://code.google.com/archive/p/marisa-trie/

ⓒ 2018 Information Processing Society of Japan

表 2 実験結果

	C_1			C_2			C_3		
	圧縮率 [%]	Lookup [μs/str]	Access [μs/str]	圧縮率 [%]	Lookup [μs/str]	Access [μs/str]	圧縮率 [%]	Lookup [μs/str]	Access [μs/str]
DAA	61.3	**0.51**	3.73	11.3	1.11	4.57	31.7	**0.77**	5.72
DAA-C	56.8	33.94	3.62	10.5	28.56	4.58	28.4	34.23	5.59
DAA-L	**51.4**	**0.51**	29.32	**10.0**	1.15	24.41	**26.2**	0.86	28.31
DAA-w	85.9	0.52	**3.00**	14.0	**1.09**	**4.10**	50.5	0.86	**4.48**
XCDA	53.0	**0.44**	**0.70**	18.1	**1.33**	**1.81**	36.2	**0.90**	**1.28**
CentRP	32.4	0.96	1.11	11.8	1.72	2.15	16.9	1.59	1.82
Marisa	**26.1**	1.10	1.01	**7.2**	3.08	3.06	**14.0**	1.69	1.69

ずれの手法よりも大きいが，C_2 では XCDA,CentRP よりも小さく，C_3 では XCDA より小さい結果が得られた．

Lookup について，提案手法のうち，C-WORDS を用いない実装の Lookup 時間は，C_1 では約 67 倍，C_2 では約 26 倍，C_3 では約 44 倍に増加しており，C-WORDS の導入による Lookup の高速化が確認できる．他のデータ構造と比較した場合，C_1 では CentRP,Marisa より高速かつ C_2, C_3 ではいずれのデータ構造よりも高速であり，提案手法は Lookup に秀でたデータ構造であると言える．

Access について，ELD,BRO を用いない実装の Access 時間は，C_1 では約 8 倍，C_2, C_3 では約 5 倍に増加しており，ELD,BRO の導入による Access の高速化が確認できる．しかし，他のデータ構造における Access の時間計算量は入力文字列長 m のみに依存し $O(m)$ であるのに対し，提案手法は m と DFA の各状態が持つ遷移数の平均 b に依存し $O(mb)$ であるため，提案手法の Access に理論的な課題が見られる．

6. おわりに

本稿では，圧縮文字列辞書を DFA により実現し，ダブル配列オートマトンをベースとしたデータ構造による実装と，実装に伴う圧縮手法を提案した．DFA での辞書の実現には，DFA の各遷移に対応するワード数を保存する必要があり，ワード数の効率的表現を目指した圧縮手法に高い効果が見られた．また，ダブル配列オートマトンをベースとした提案手法は Lookup 時間に優れ，同程度の速度を実現する他の手法と比較しても同等以上の空間効率を実現した．辞書構造において Lookup は最も実用的な操作であり，圧縮文字列辞書の多くの用途において Lookup の高速性が重要視されている．提案手法は，少なくとも辞書構造における最も重要な機能に対し有効な手法であると言える．一方で Access の実行は低速であり，圧縮文字列辞書の幅広い用途としては十分な機能を満たすことができていない．辞書 DFA の理論的欠点とその表現方法の両面における改善は今後の課題である．

また本稿では辞書の構築時間に対する評価を与えていないため，構築時間に対する実験と評価も今後の予定として挙げられる．

参考文献

[1] Martnez-Prieto, M. A., Brisaboa, N., Cnovas, R., Claude, F. and Navarro, G.: Practical compressed string dictionaries, *Information Systems*, Vol. 56, pp. 73 – 108 (online), DOI: https://doi.org/10.1016/j.is.2015.08.008 (2016).

[2] 矢田 晋：Prefix/Patricia Trie の入れ子による辞書圧縮，言語処理学会第 17 回年次大会発表論文集，pp. 576–578 (2011).

[3] Grossi, R. and Ottaviano, G.: Fast Compressed Tries throwgh Path Decompositions, *Journal of Experimental Algorithmics (JEA)*, Vol. 19, pp. 3–4 (2014).

[4] Kanda, S., Morita, K. and Fuketa, M.: Compressed double-array tries for string dictionaries supporting fast lookup, *Knowledge and Information Systems*, Vol. 51, No. 3, pp. 1023–1042 (2017).

[5] Hopcroft, J. E., Ullman, J. and Motowani, R.: オートマトン・言語理論・計算論 (2003).

[6] 前田敦司，水島宏太：オートマトンの圧縮配列表現と言語処理系への応用，プログラミングシンポジウム，Vol. 49, pp. 49–54 (2008).

[7] Martn-Vide, C.: Scientific Applications of Language Methods (2011).

[8] González, R., Grabowski, S., Mäkinen, V. and Navarro, G.: Practical implementation of rank and select queries, *Poster Proceedings of the 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pp. 27–38 (2005).

[9] 神田峻介，森田和宏，泓田正雄：文字列辞書を用いた効率的な文字列辞書圧縮の検討と評価，日本データベース学会和文論文誌，Vol. 16-J, No. 7 (2018).

[10] 松本拓真，神田峻介，森田和宏，泓田正雄：ダブル配列オートマトンの圧縮手法，*DEIM Forum*, No. P5-1 (2018).

[11] 矢田 晋，田村雅浩，森田和宏，泓田正雄，青江順一：ダブル配列による動的辞書の構成と評価，全国大会講演論文集，Vol. 71, No. ソフトウェア科学・工学，pp. 263–264 (2009).

ⓒ 2018 Information Processing Society of Japan