# ダブル配列オートマトンの圧縮手法

松本　拓真†　　神田　峻介††　　森田　和宏††　　泓田　正雄††

† 徳島大学工学部知能情報工学科　〒 770–8506 徳島県徳島市南常三島町 2–1
†† 徳島大学大学院先端技術科学教育部　〒 770–8506 徳島県徳島市南常三島町 2–1
E-mail: †{tkm.matsumoto,shnsk.knd}@gmail.com, ††{kam,fuketa}@is.tokushima-u.ac.jp

**あらまし**　決定性有限オートマトンは，自然言語処理や情報検索における単語辞書の実現などに広く用いられている．そしてオートマトンを効率よく表現する手法は様々に工夫されている．効率的なオートマトンの表現方法として，前田らの提案するダブル配列表現がある．ダブル配列表現は高速な検索を実現するが，ポインタ列に起因する記憶量の大きさが課題である．これを解決するため，本稿ではダブル配列オートマトンの圧縮手法を提案する．

**キーワード**　辞書，データ構造，オートマトン，ダブル配列

## 1. はじめに

文字列集合を扱う代表的な有向グラフに，決定性有限オートマトンがある．オートマトンは，文字列の接頭辞と接尾辞を併合することによりコンパクトに文字列集合を保存でき，自然言語処理や情報検索などにおける単語辞書の実現 [1] [2] などに広く利用されている．オートマトンを効率よく表現する手法は様々に工夫されており，記憶量，検索速度などの性能は各手法により特徴がある [3] [4] [5].

オートマトンの表現手法の一つに，前田らの提案するダブル配列オートマトン [5] がある．ダブル配列は 2 つの一次元配列でオートマトンを表現し，高速な検索を実現する．一方で，ポインタ列に起因する記憶量の大きさが課題となっている．ダブル配列オートマトンは，青江の提案するダブル配列トライ [6] を改良したものであるが，ダブル配列トライに対する圧縮結果は多く示されている一方，オートマトンを表現した前田の手法への圧縮結果は示されていない．

そこで本稿では，ダブル配列オートマトンの圧縮手法として以下の 2 つの手法を提案する．

（1） ダブル配列の要素サイズの圧縮 (節 3.)
（2） オートマトンの遷移数削減による圧縮 (節 4.)

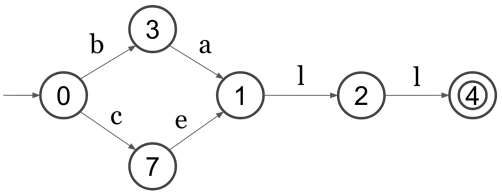
## 2. オートマトンのダブル配列表現

ダブル配列は 2 つの 1 次元配列 NEXT と CHECK を用いてオートマトンを表現し，配列の各要素はオートマトンの遷移に対応する．ダブル配列における状態 $s$ から状態 $t$ への文字 $c$ による遷移を次式により表す．

$$\begin{cases} e \leftarrow s \oplus c \\ \mathrm{CHECK}[e] = c \\ t \leftarrow \mathrm{NEXT}[e] \end{cases} \tag{1}$$

キー集合 $K = \{\text{“}ball\text{”}, \text{“}cell\text{”}\}$ に対するダブル配列オートマトンを図 1 に示す．図において，アルファベット集合 {‘a’,‘b’,‘c’,‘e’,‘l’} は，{0,1,2,3,4} の整数として扱われる．

ダブル配列は式 (1) により定数時間で状態間の遷移を実行できる．検索時間は入力文字列の長さ $k$ にのみ依存し $O(k)$ である．記憶量について，NEXT は状態番号を格納し，配列長を $n$ とすると各要素 $\lceil \log n \rceil$ ビットを要する．CHECK 配列は文字を格納するため，アルファベットサイズを $\sigma$ とすると各要素 $\lceil \log \sigma \rceil$ ビットを要する．則ちダブル配列はオートマトンの表現に，$n(\lceil \log n \rceil + \lceil \log \sigma \rceil)$ ビットを要する．




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| NEXT | 0 | 3 | 7 | 1 | 1 | 2 | 4 |
| CHECK | | b | c | a | e | l | l |

図 1　キー集合 $K$ に対するダブル配列オートマトン

## 3. 要素圧縮

ダブル配列の NEXT と CHECK の記憶量はそれぞれ各要素 $\lceil \log n \rceil$, $\lceil \log \sigma \rceil$ ビットであるが，基本的に $n \gg \sigma$ であり，NEXT の記憶量がボトルネックである．この NEXT について Kanda ら [7] と同様の手順で圧縮を試みる．圧縮の手順を以下に示す．

（1） NEXT 配列の値をより小さい値で表現できるように遷移式を変形する．
（2） 透過的符号化により，これらの値をコンパクトに表現する．

**手順 1**

式 (1) を変形する．新たに用いる配列を NEXT’ とし，NEXT’ を用いた状態 $s$ から $t$ への遷移を次式により表す．

$$\begin{cases} e \leftarrow s \oplus c \\ \mathrm{CHECK}[e] = c \\ t \leftarrow e \oplus \mathrm{NEXT'}[e] \end{cases} \quad (2)$$

NEXT' は NEXT'$[e] \leftarrow e \oplus$ NEXT$[e]$ の値で構成される配列である．$e \oplus$ NEXT'$[e] = e \oplus e \oplus$ NEXT$[e] =$ NEXT$[e]$ より，式 (2) が式 (1) の示す遷移と同様の計算を示していることが分かる．

NEXT' の値の大きさについて，$(e \oplus \mathrm{NEXT}[e]) < 2^k$ であった場合，NEXT'$[e]$ は $k$ ビットで表される．ダブル配列では式 (1) を満たす限り，NEXT の値を自由に決定することが出来る．$e \oplus$ NEXT$[e]$ が小さくなるように構築することにより，NEXT' を全体的に小さい値で構成することが出来る．

**手順 2**

Directry addressable codes (DAC) [8] を用いる．DAC は，整数値列をランダムアクセスが可能なまま，コンパクトに表現する手法である．整数値列をバイト列に分割し，値の表現に必要なバイトのみを保存することで，整数値列の記憶量を削減する．

例として，整数値配列 $P$ に対する DAC 表現を図 2 に示す．$P[i]$ の表現に必要なバイト数を $k_i$ とし，$P[i]$ をバイト列に分割して $p_{(i,k_i)}, ..., p_{(i,2)}, p_{(i,1)}$ と表す．また，整数 $x$ の 2 進数表現を $(x)_2$ と表すことにする．$P[i] = 277$ の場合，$(277)_2 =$ 1 00010101 より，$p_{(i,2)} =$ 00000001, $p_{(i,1)} =$ 00010101 となる．$A_j$ は $j$ 番目のバイト値が存在する全ての $P[i]$ における，$j$ 番目のバイト値を切り詰めて保存したバイト列である．また $B_j$ はビット列で，$A_j[i]$ のバイト値が $A_{j+1}$ に存在する場合に，$B_j[i] = 1$ を与える．

ここで，全ての $B_j$ に対して Rank 辞書 [9] を構築する．Rank 辞書とは，ビット列 $B$ に対して $o(|B|)$ ビットを追加することにより以下の操作を定数時間で可能にする補助データ構造のことである．

- Rank$(B, i)$ : $B[0...i-1]$ 中の 1 の数を返す

この Rank 辞書を用いて，$P[i]$ を以下の様に復元する．まず，$P[i]$ を表すための $A_j$ における添字を $i_1, i_2, ..., i_{k_i}$ と表すことにする．つまり，$A_1[i_1] = p_{(i,1)}, A_2[i_2] = p_{(i,2)}, A_{k_i}[i_{k_i}] = p_{(i,k_i)}$ となる．次に，$i_1 = i$ とし，$i_2, ..., i_{k_i}$ を式 3 により順に算出する．

$$i_{j+1} = \mathrm{Rank}(B_j, i_j)\ (B_j[i_j] = 1) \quad (3)$$

$B_j[i_j] = 0$ の場合，$j = k_i$ であり，$A_j[i_j]$ が $P[i]$ を表現する最後のバイトとなる．$l = \lceil |(P[i])_2| / 8 \rceil$ とすると，$P[i]$ の復元には $l - 1$ 回の Rank 計算を要する．つまり $l = 1$ の場合，$P[i]$ は 1 バイトで表現でき，高速に復元することが出来る．

即ち，NEXT' を DAC で表現することにより，NEXT' の値が全体的に小さい値で構成されているほど，小さい記憶量で表現でき，かつ復元時間の増加を抑えることが出来る．

## 4. 遷移数削減

ダブル配列の要素数は，表現したオートマトンの遷移数に対応している．そこで，文字列による遷移を許容し，前後に分岐を持たず受理状態でない状態を削除する．これによりオートマトンの遷移数が削減され，ダブル配列の配列長が短くなる．図 1 における削除可能な状態は 3,7,2 である．これらの状態を削減したキー集合 $K$ に対するオートマトンを図 3 に示す．

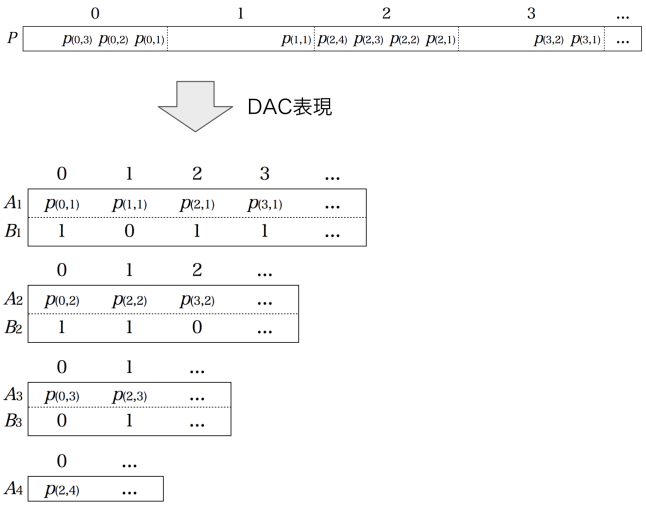



図 2　整数値配列 $P$ に対する DAC 表現

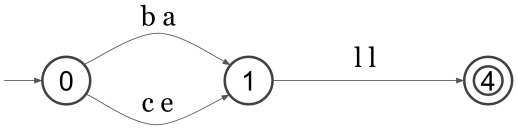



図 3　キー集合 $K$ に対する文字列で遷移するオートマトン

ここで，CHECK は固定長配列なので，一つの要素に文字列を与えることは出来ない．そこで，矢田 [10] や神田ら [11] の手法を応用し，以下の手順で文字列による遷移をダブル配列で表現する．

（1） 辞書符号化により文字列に整数値の ID を与える．

（2） CHECK にその ID を格納する．

**手順 1**

文字列を格納する配列 STR を導入する．遷移に文字列が与えられた場合，それらの文字列に終端文字 '#' を付加し，配列 STR に連続して格納する．そして文字列の開始位置の添字をその文字列の ID とする．この方法では，開始位置の添字をずらすことで文字列を部分的に取り出すことが出来るため，他の文字列の後方に一致する文字列を併合することが出来る．

**手順 2**

CHECK$[e]$ が文字か ID かを区別するため，ビット列 FLAG を導入する．CHECK$[e]$ が ID であった場合，FLAG$[e] = 1$ とする．CHECK に文字と ID を格納できるように記憶量を設定した場合，各要素 $max(\lceil \log \sigma \rceil, \lceil \log |\mathrm{STR}| \rceil)$ ビットを要することになる．しかし基本的に $\sigma < |\mathrm{STR}|$ であり，文字を格納する要素では $\lceil \log |\mathrm{STR}| \rceil - \lceil \log \sigma \rceil$ ビットの無駄が生じることになる．そのため，まず CHECK は各要素 $\lceil \log \sigma \rceil$ ビットの配列とし，ID の上位 $\lceil \log \sigma \rceil$ ビットを CHECK に格納する．そして，残りの下位 $\lceil \log |\mathrm{STR}| \rceil - \lceil \log \sigma \rceil$ ビットを格納する配列

FLOW を導入する．また，ビット配列 FLAG に対して Rank 辞書を構築する．遷移 $e$ における ID を $x$ とすると，FLAG，CHECK，FLOW の値はそれぞれ以下の様になる．

$$\begin{aligned} \mathrm{FLAG}[e] &= 1 \\ \mathrm{CHECK}[e] &= x \bmod 2^{\lceil \log \sigma \rceil} \\ \mathrm{FLOW}[\mathrm{Rank}(\mathrm{FLAG}, e)] &= \lfloor x / 2^{\lceil \log \sigma \rceil} \rfloor \end{aligned}$$

## 5. 評　　価

本節では，節 2. で紹介したダブル配列に対し，節 3. で紹介した要素圧縮と，節 4. で紹介した遷移数削減を適応した場合それぞれの性能に関して評価を与える．

### 5.1 実 験 設 定

辞書の実装言語は C++であり，Intel Core i7 4 GHz CPU を搭載した PC 上で比較を行った．実験では，キー集合に対しダブル配列を構築したときの記憶量と，1 キーあたりの検索時間を求めた．

**コーパス**　辞書の構築に用いたコーパスを以下に示し，詳細を表 1 に示す．

- $C_1$ : 日本語版 Wikipedia の見出し集合（2015 年 1 月時点）(注1)
- $C_2$ : インドシナ諸国のドメイン上でクロールし得られた URL 集合 (注2)

表 1　実験に用いたコーパス

| | サイズ [MB] | 単語数 | 平均長 |
|---|---|---|---|
| $C_1$ | 32.3 | 1,518,205 | 22.31 |
| $C_2$ | 612.9 | 7,414,866 | 86.68 |

### 5.2 結果と評価

実験結果を図 4,5 に示す．図においてラベルが表す結果を以下に示す．

- D : オリジナルのダブル配列
- N : D に対し要素圧縮を行なった結果
- E : D に対し遷移数削減を行った結果

平均検索時間を計測するために，コーパスからランダムに抽出した 100 万個の文字列を用いた．それぞれ，10 回の試行から得られた結果の平均である．

**要素圧縮の結果に対する評価**

$C_1$ では，検索時間は 2.10 倍と大きく増加したが，記憶量は 0.71 倍まで削減された．一方 $C_2$ では，検索時間は 1.64 倍で，記憶量は 0.52 倍まで削減された．キー長の平均が長いコーパスでは，ダブル配列の NEXT' の値が全体的に小さくなるため，$C_2$ に対してより良い圧縮結果が得られた．時間増加について，圧縮に用いた DAC では，表現する値が大きいほど値の復元速度が低下することが原因である．

(注1) : https://dumps.wikimedia.org/jawiki/

(注2) : http://data.law.di.unimi.it/webdata/indochina-2004/indochina-2004.urls.gz

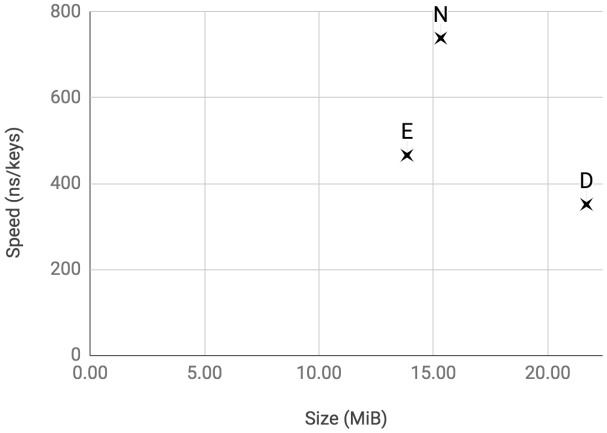



図 4　$C_1$ に対するメモリと検索速度のトレードオフ

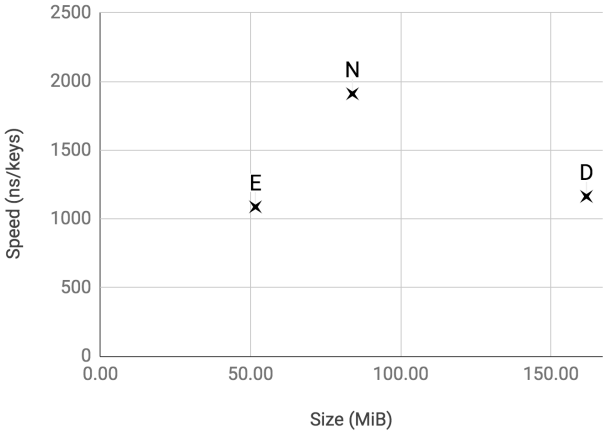



図 5　$C_2$ に対するメモリと検索速度のトレードオフ

**遷移数削減の結果に対する評価**

記憶量について，キー長の平均が長く，削減可能な遷移の数が多いコーパスの削減率が高くなった．中でも $C_2$ に対して特に有効であり，0.32 倍まで削減された．注目すべきは検索時間で，$C_2$ において 0.93 倍であり，時間効率を悪化させずに圧縮を実現できている．即ちダブル配列の特徴である高速性を維持した圧縮手法であると考えられる．実験では，いずれのコーパスでも，要素圧縮の結果と比較し時間効率, 空間効率の両面に優れていた．

## 6. まとめと今後の課題

本稿では，ダブル配列オートマトンの 2 つの圧縮手法を提案した．要素圧縮の結果は，速度とのトレードオフによりメモリ圧縮を実現できた．遷移数削減の結果は，要素圧縮よりメモリと速度の両面に優れており，ダブル配列の高速性を維持できる圧縮手法であった．

今後の課題として，要素圧縮に用いた DAC では，値の復元における速度損失の影響が大きいため，DAC に代わる値のコンパクト表現を可能とする手法が望まれる．また，遷移数削減時の文字列符号の格納に関して，ダブル配列の構造特性を利用

した改善の見込みがある．これを実装し，検証を行っていく予定である．

## 文　　献


[1] Jan Daciuk, Jakub Piskorski, and Strahil Ristov. Natural language dictionaries implemented as finite automata. In Carlos Martín-Vide, editor, *Scientific applications of language methods*, chapter 4, pp. 133–204. World Scientific, 2010.

[2] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. Apache lucene 4. In *Proceedings of the SIGIR 2012 workshop on Open Source Information Retrieval*, p. 17, 2012.

[3] Jan Daciuk. Experiments with automata compression. In *Proceedings of the International Conference on Implementation and Application of Automata (CIAA)*, pp. 105–112, 2000.

[4] Jan Daciuk and Dawid Weiss. Smaller representation of finite state automata. *Theoretical Computer Science*, Vol. 450, pp. 10–21, 2012.

[5] 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.

[6] Jun'ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077, 1989.

[7] Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowledge and Information Systems*, Vol. 51, No. 3, pp. 1023–1042, 2017.

[8] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, Vol. 49, No. 1, pp. 392–404, 2013.

[9] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings of the 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pp. 27–38, 2005.

[10] 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.

[11] 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, p. Article 7, 2018.