Vol.2015-DBS-161 No.10
Vol.2015-IFAT-119 No.10
2015/8/6

# トライにおける逆方向遷移可能かつコンパクトな配列構造

神田 峻介[1,a)]　泓田 正雄[1]　森田 和宏[1]　青江 順一[1]

概要：トライと呼ばれる順序木を効率的に表現するデータ構造として，高速な検索を提供するダブル配列がある．また，データの大規模化に伴いコンパクト性が重視される背景に応じて，様々なダブル配列の圧縮表現が提案されてきた．しかし，これらの圧縮表現は，トライにおける順方向の遷移（親から子）のみを提供し，逆方向の遷移（子から親）を提供していないため，結果としてダブル配列における逆引きや動的更新を犠牲にしている．本論文では，逆方向遷移を可能としたコンパクトな配列構造を提案する．記憶量について，ダブル配列の約 36%でトライを表現可能なことが実験により確認されている．

キーワード：トライ，ダブル配列，圧縮表現，情報検索

# A Compact Array Structure with Reverse Traversal in a Trie

Shunsuke Kanda[1,a)]　Masao Fuketa[1]　Kazuhiro Morita[1]　Jun-ichi Aoe[1]

***Abstract:*** A double-array efficiently represents an ordered tree called “trie”, and provides fast retrieval in the trie. Recently, many compact representations of the double-array have been proposed because a compact data structure is important in big data. The double-array provides inorder (from a parent to a child) and reverse (from a child to a parent) traversals, but the compact representations provide only the inorder traversal. Therefore, the compact representations don’t have a dynamic update and a reverse lookup in the double-array. This paper proposes a compact array structure with the reverse traversal. In addition, experimental results show that the new structure represents the trie with 36% of the double-array size.

***Keywords:*** Trie, Double-array, Compact representation, Information retrieval

## 1. はじめに

あらゆるデータが文字列として表されている現代において，文字列を扱うためのアルゴリズムとデータ構造は重要である．その中でも，枝に文字を付随した順序木であるトライ（Trie）[1] は，文字列内のキー（単語）を効率的に管理するため，情報検索における索引を始め幅広く用いられている．このトライを効率的に表現するデータ構造として，青江によって提案されたダブル配列（Double-array，DA）[2] がある．ダブル配列は，検索の高速性に秀でたデータ構造であり，その検索速度は入力文字列の長さのみに依存する．また，トライにおける逆引き（キーの復元）や動的更新（キーの追加や削除）などの機能も持ち合わせており，様々なシステムに応用できる．

一方で，高度情報化社会の進展に伴うデータの大規模化に際し，トライのコンパクトな表現が重視されてきている．ダブル配列においても様々な圧縮表現が提案されており，その前身となるのが圧縮ダブル配列（Compact double-array，CDA）[3] である．圧縮ダブル配列は，検索の高速性を維持したまま，配列の各要素が必要とする表現領域を削減したデータ構造であり，実際の記憶量はダブル配列と比べ約 62.5%である．しかし，トライにおける順方向の遷移（親から子）のみを提供し，ダブル配列が可能としていた逆方向の遷移（子から親）を損うため，結果として，逆引きや動的更新などの機能を犠牲にしている．トライに対し，頻繁な更新などを必要としないシステムでは，圧縮ダブル配列でも有用であるが，今後，更なるデータの

[1] 徳島大学大学院先端技術科学教育部
Department of Information Science and Intelligent Systems, Tokushima University
[a)] shnsk.knd@gmail.com

Vol.2015-DBS-161 No.10
Vol.2015-IFAT-119 No.10
2015/8/6

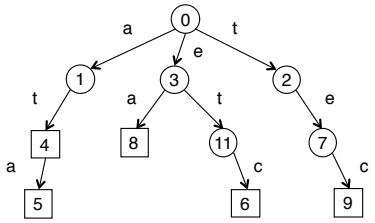


図 1　キー集合 $K$ に対するトライ.
**Fig. 1**　A trie for keyword set $K$.

大規模化における構築コストの増大や高度な解析の要求を考慮したとき，これらの機能は不可欠である．

本論文では，逆方向遷移を損なわないダブル配列の圧縮表現を提案し，実験により実環境での有効性を示す．結果として，提案手法はダブル配列の約 36%，圧縮ダブル配列の約 58%の記憶量でトライを表現でき，かつダブル配列の特徴を引き継いでいるため，他のトライ表現と比べても充分に高速である．また，逆方向遷移が可能なため，ダブル配列と同様の手順で動的更新も可能である．

## 2. トライとその表現法

### 2.1 トライ

キー集合における接頭辞を併合圧縮し，枝に文字を付随することで構築される順序木がトライである．根から葉への経路上のラベルを連結した文字列がキーに対応する．但し，キーが他のキーの接頭辞である場合は，内部節点がキーの終端となる．そのため本研究では，配列 TERM を用いて $\mathrm{TERM}[s] \in \{0,1\}$ とすることで，各節点 $s$ がキーの終端であるかを識別する．例として，キー集合 $K = \{$“at”, “ata”, “ea”, “etc”, “tec”$\}$ に対するトライを図 **1** に示す．図において四角の節点が，キーの終端を意味している．トライの表現法について述べる上で，本論文では実際的なトライを想定し，以下を定義する．

**定義 1**　あらゆるマルチバイト文字もバイト文字列として扱われる場合を想定し，アルファベット集合 $\{0,1,\cdots,255\}$ 上の文字を枝に付随した，節点数 $n$ のトライを扱う．このとき，各文字は $\log 256 = 8$ ビット [*1] で表される．また，語長 32 ビットの計算機上において，節点数 $n$ は $2^{32}$ に遠く及ばないとし，節点を表現するための整数は 32 ビットで表されるものとする．例えば，英語版 Wikipedia[*2] の見出し語のような大規模なキー集合に対してトライを構築した場合でも，$n = 99,442,676 < 2^{27}$ なので，この定義は実際的といえる．

### 2.2 ダブル配列

ダブル配列は，2 つの一次元配列 BASE，CHECK を用いてトライの節点を表現するデータ構造であり，任意の内部節点 $s$ から出ている文字ラベル $c$ の枝が節点 $t$ を指す状態に対し，次式を満足する．

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

図 **2**　図 1 のトライに対するダブル配列表現.
**Fig. 2**　Double-array representation of the trie in Fig. 1.

$$\mathrm{BASE}[s] \oplus c = t,\ \mathrm{CHECK}[t] = s \qquad (1)$$

すなわち，1 回の排他的論理和演算 [*3] と 1 回の等価演算で子ノードへの遷移をおこなえるため，理論的にも実際的にも検索は極めて高速である．子から親への逆方向の遷移も，CHECK が親を直接示しているため容易におこなえ，リンクトライ [4] などの逆方向遷移を必要とするデータ構造にも適している．また，ダブル配列の動的更新では節点の挿入において，任意の節点からその親を特定する必要があるため，CHECK が親を示す役割は重要である．

記憶量に関しては，BASE，CHECK 値ともに節点を表現するために 32 ビット必要とし，全体の記憶量は $64n$ ビットである．厳密には，式 (1) を満足するために発生する空要素数についても考えるべきだが，基本的に $n$ と比べると無視できるほど小さい値のため，本論文の理論的評価では空要素数は 0 と仮定する．TERM については，BASE か CHECK どちらかの記憶領域から 1 ビット得ることで表されるのが一般的である．

**例 1**　図 1 のトライのダブル配列表現を図 **2** に示す．但し，文字集合 $\{$‘a’, ‘c’, ‘e’, ‘t’$\}$ は，$\{0,1,2,3\}$ の整数として扱われている．図 2 において，節点 3 から 11 へ文字 ‘t’ による遷移は，$\mathrm{BASE}[3] \oplus$ ‘t’ $= 8 \oplus 3 = 11$，$\mathrm{CHECK}[11] = 3$ によって満足する．

### 2.3 ダブル配列における圧縮表現

データのコンパクト性が重視される背景に応じて，様々なダブル配列の圧縮表現が提案されてきた．これらの圧縮表現の前身となるのが，矢田らによって提案された圧縮ダブル配列である．任意の内部節点 $s$ から出ている文字ラベル $c$ の枝が節点 $t$ を指す状態に対し，次式を満足する．

$$\mathrm{BASE}[s] \oplus c = t,\ \mathrm{CHECK}[t] = c \qquad (2)$$

ダブル配列と異なる点は，CHECK により親ではなく遷移文字を照合することで，遷移の存在を識別している点である．但し，複数の節点から同じ節点への遷移が成立しないように，内部節点の BASE 値の重複は禁止される．

圧縮ダブル配列では，CHECK に文字が格納されること

[*1] 本論文では，対数の底は 2 とする．
[*2] http://dumps.wikimedia.org/enwiki/20150205/enwiki-20150205-all-titles-in-ns0.gz
[*3] 一般的には和演算 (+) が用いられているが，排他的論理和演算 ($\oplus$) を代わりに用いることもできる [3].

により，CHECK の各要素は 8 ビットで表されるため，全体の記憶量は $40n$ ビットに軽減される．一方で，CHECK により親の特定ができないため，以降の圧縮ダブル配列を前身とする圧縮表現 [5], [6] も含め，動的更新などの機能を犠牲している．

### 2.4 簡潔データ構造

簡潔データ構造を用いれば，定数時間で遷移を実現しながら，情報理論的に最適な記憶量で順序木を表現することができる．トライに関しては，文字ラベルを別領域に保存しておくことで，簡潔データ構造によって表現される．特に，LOUDS（Level-Order Unary Degree Sequence）[7] と呼ばれるデータ構造は，表現の容易さや適応性からトライの表現によく用いられている．LOUDS がトライの節点と文字を表現するために必要な記憶量は，標準的な実装 [*4] であれば約 $2.75n$ と $8n$ ビットであり，TERM のための $n$ ビットを加えたとしても，この記憶量は圧縮ダブル配列の $40n$ ビットと比べてかなり小さい．しかし，遷移において余分なノードの探索と多くのビット演算を要するため，ダブル配列と比べると低速であり，実際，その速度差は 10 倍以上である [6]．また，逆方向遷移はおこなえるが，動的更新をおこなうことはできない．

一方で，定兼らによって提案された手法 [8] を用いれば，BP（Balanced Parentheses）[9] などの括弧表現による順序木のための簡潔データ構造を容易に表現することができる．更に，動的更新も可能となり，実用面においても高い性能を発揮する．しかし，子ノードの探索や枝の文字ラベルの取得においては LOUDS の方が高速である [10]．

## 3. 逆方向遷移可能かつコンパクトな配列構造

本節では，ダブル配列をコンパクトな配列構造に変形することによって，逆方向遷移可能な圧縮表現を提案する．提案手法による配列構造は，ダブル配列と同様の手順で動的更新なども実現できる．初めに，提案手法を説明する上で必要な定義及び定理を与える．次に，データ構造及びコンパクト化に向けた構築手順について説明する．

### 3.1 準備

提案手法において重要となるのが，ダブル配列のブロック分割である．配列のブロック分割に関して，以下の諸定義を与える．

**定義 2** 配列を長さ $x$ のブロックに分割したとき，要素 $s$ が属するブロックの番地を $blk(s) = \lfloor s/x \rfloor$ と表す．但し，$x$ は 2 の累乗とする．そして，整数配列 $A$ に対し，$blk(A[s]) \neq blk(s)$ なる値を $out(A, s) = \text{TRUE}$ によって表す．但し，$A[s] = \text{NIL}$ なる値に関しては，$out(A, s) = \text{FALSE}$ と定義する．また，BASE と CHECK において，$out$ によって TRUE が得られる要素の割合をそれぞれ $p_b$，$p_c$ と表す $(0 \leq p_b \leq 1,\ 0 \leq p_c \leq 1)$．

[*4] Tx: Succinct Trie Data structure. https://code.google.com/p/tx-trie/


図 3 図 2 のダブル配列に対する提案手法による配列表現．
**Fig. 3** New array representation of the double-array in Fig. 2.

**例 2** 図 **3**(a) のダブル配列は，図 2 を長さ $x = 4$ のブロックに分割した結果を表す．このダブル配列において，$blk(7) = 1$, $blk(\text{BASE}[7]) = 2$ であるので，$out(\text{BASE}, 7) = \text{TRUE}$ となり，このように $out$ によって TRUE が得られる値は網掛けで示している．一方，$blk(5) = 1$, $blk(\text{CHECK}[5]) = 1$ であるので，$out(\text{CHECK}, 5) = \text{FALSE}$ である．また，$p_b = 5/12 = 0.41$，$p_c = 6/12 = 0.50$ である．

このとき，$out(A, s) = \text{FALSE}$ となる $A[s]$ について，以下の定理が得られる．

**定理 1** $x$ が 2 の累乗のとき，$out(A, s) = \text{FALSE}$ となる $A[s]$ について，$A[s] \oplus s = s'$ は $\log x$ ビットで表現可能な値である．そのため，$s \oplus s' = A[s]$ なことから，$\log x$ ビットで $s'$ を保存しておけば $A[s]$ は復元できる．但し，$A[s] = \text{NIL}$ は例外とする．

［証明］ $x$ が 2 の累乗の場合，$blk(s) = \lfloor s/x \rfloor$ は，$s$ を右へ $\log x$ ビットシフトしたことと同義である．すなわち，任意の整数値 $i, j$ に対し，$blk(i) = blk(j)$ となる $i$ と $j$ の下位 $\log x$ ビット以外の符号は同一となる．$out(A, s) = \text{FALSE}$ においては，$blk(A[s]) = blk(s)$ であるので，$A[s]$ と $s$ の下位 $\log x$ ビット以外の符号は同一であり，$A[s] \oplus s = s'$ の下位 $\log x$ ビット以外の符号は全て 0 になる．そのため，$s'$ の表現には $\log x$ ビットで充分であり，$s'$ さえ保存しておけば，$s \oplus s'$ で $A[s]$ を復元できる． □

表 1　各操作におけるダブル配列と提案手法の対応.
**Table 1**　Correspondences for each operation in the double-array and the new method.

| | ダブル配列 | 提案手法 |
|---|---|---|
| BASE 値の取得 | BASE[$s$] | **if** BBV[$s$] = 1 **then** LBASE$_{blk s}$[SBASE[$s$]] **else** $s \oplus$ SBASE[$s$] |
| CHECK 値の取得 | CHECK[$s$] | **if** CBV[$s$] = 1 **then** LCHECK$_{blk s}$[SCHECK[$s$]] **else** $s \oplus$ SCHECK[$s$] |
| キーの終端の識別 | TERM[$s$] = 1 | TERM[$s$] = 1 |
| 葉ノードの識別 | BASE[$s$] = NIL **and** CHECK[$s$] $\neq$ NIL | TERM[$s$] = 1 **and** LE[$s$] = 1 |
| 空要素の識別 | BASE[$s$] = NIL **and** CHECK[$s$] = NIL | TERM[$s$] = 0 **and** LE[$s$] = 1 |

### 3.2　データ構造

提案手法では定理 1 を用いて，*out* によって FALSE が得られる BASE，CHECK 値を $\log x$ ビットで管理し，TRUE が得られる値のみを 32 ビットで管理する．例として，図 2 のダブル配列を提案手法により変形した結果を図 3(b) に示す．図における LBASE，LCHECK が 32 ビット表される配列であり，SBASE，SCHECK が $\log x$ ビット表される配列である．また，BBV，CBV は各要素に対し，*out* が TRUE か FALSE かを識別するためのビット配列であり，LE は各要素が葉か空かを識別するためのビット配列である．提案手法ではこれらの配列を用いることで，変形前のダブル配列を小容量でかつ高速に復元する．そのため，遷移式は変わらず式 (1) を用いることができ，検索も高速である．

変形手順の説明も兼ねて，各配列の詳しい構成を以下に示し，**表 1** に各復元方法をまとめた結果を示す．

**BBV**，**CBV の構成**　BASE/CHECK Bit Vector を意味する．BBV[$s$] $\in \{0, 1\}$ とすることで，BBV[$s$] = 0 であれば $out$(BASE, $s$) = FALSE であると識別する．BBV[$s$] = 1 であれば $out$(BASE, $s$) = TRUE であると識別する．CBV についても，CHECK に対し同様の構成である．

**LBASE**，**LCHECK の構成**　Large-BASE/CHECK を意味する．ダブル配列の各ブロックに応じて LBASE 配列を定義し，$out$(BASE, $s$) = TRUE なる BASE 値を LBASE$_{blk(s)}$ に格納する．このとき，LBASE$_{blk(s)}$ の要素数は $x$ 以下である．LBASE 値は BASE 値と同じく 32 ビットで表され，記憶量は $32n \cdot p_b$ ビットである．LCHECK 配列も CHECK に対し同様の構成であり，記憶量は $32n \cdot p_c$ ビットである．

**SBASE**，**SCHECK の構成**　Small-BASE/CHECK を意味する．定理 1 を用いて，各要素が $\log x$ ビットで表される配列であり，ダブル配列と同様に各要素は各節点に対応する．$out$(BASE, $s$) = FALSE なる BASE 値において，$s \oplus$BASE[$s$] を SBASE[$s$] に保存しておき，$s \oplus$SBASE[$s$] = BASE[$s$] によって復元する．また，$out$(BASE, $s$) = TRUE なる SBASE[$s$] は，BASE[$s$] = LBASE$_{blk(s)}$[$i$] なる $i$ を SBASE[$s$] に格納しておくことで，LBASE の参照に用いられる．LBASE$_{blk(s)}$ の要素数は $x$ 以下なので，$i$ は $\log x$ ビットで表せる．SCHECK についても，CHECK に対し同様の構成である．

**TERM**，**LE の構成**　提案手法では SBASE，SCHECK の表現領域を全て使用するため，NIL によって葉や空要素を表現することができない．そのため，ビット配列 LE（LEAF or Empty）と TERM を用いて，ダブル配列におけるキーの終端，葉，空要素を表現する．TERM については従来と同様であり，キーの終端であれば 1 を設定する．LE については，葉もしくは空要素であれば 1 を設定する．LE[$s$] = 1 かつ TERM[$s$] = 1 であれば，終端であることから節点 $s$ が葉であることがわかる．LE[$s$] = 1 かつ TERM[$s$] = 0 であれば，終端ではないため要素 $s$ が空であることがわかる．

以上の構成により，$p_b$ と $p_c$ が小さいほどコンパクトにダブル配列を表現することができる．また，LBASE，LCHECK の構成も非常に単純なため，動的更新による要素の追加，削除にも充分に対応できる．

**例 3**　図 3 における (b) の配列構造から (a) のダブル配列を復元する場合を考える．BASE[7] = 8 を復元するとき，BBV[7] = 1 より $out$(BASE, 7) = TRUE であることがわかるため，LBASE$_{blk(7)}$[SBASE[7]] = LBASE$_1$[0] = 8 により復元できる．CHECK[5] = 4 を復元するとき，CBV[5] = 0 より $out$(CHECK, 5) = FALSE であることがわかるため，$5 \oplus$ SCHECK[5] $= 5 \oplus 1 = 4$ により復元できる．また，TERM[5] = 1 かつ LE[5] = 1 より節点 5 が葉であることがわかり，TERM[10] = 0 かつ LE[10] = 1 より要素 10 が空であることがわかる．

### 3.3　構築手順

$p_b$，$p_c$ は LBASE，LCHECK の記憶量に大きく依存し，これらは小さい値であるほど良い．本節では，$p_b$，$p_c$ を減少させる構築手順について述べる．

一般的なダブル配列の構築手順では，節点の定義において式 (1) を満足する空要素を前方から順に利用する．一方で，矢田らが [3] で示している構築手順では，メモリキャッシュの効率化を目的し，親と子の要素を同一キャッシュライン上に配置することを優先する．すなわち，前方の空要

表 2　理論的観測の結果.
**Table 2**　Theoretical results.

| 手法 | 検索時間 | 記憶量 [ビット] |
|---|---|---|
| ダブル配列 | $O(k)$ | $64n$ |
| 圧縮ダブル配列 | $O(k)$ | $40n$ |
| 提案手法 | $O(k)$ | $20n + 32n(p_b + p_c)$ |
| LOUDS | $O(k\sigma)$ | $11.75n + 1.375$ |

素から順に利用するのではなく，親と同一のキャッシュライン上の空要素を優先して利用する．また，[3] では，式 (1) のように子の特定を排他的論理和によりおこなうことで，ラインの確認を容易にしている．

この手順を利用し，親と子の要素を同一ブロック上に配置することを優先すれば，$out$ によって FALSE が得られる BASE，CHECK 値をより多く定義できる．また，子の特定に排他的論理和を用いるとき，アルファベット集合が $\{0, 1, \cdots, \sigma - 1\}$ であれば，ブロック長 $x = 2^{\lceil \log \sigma \rceil}$ にすると効率が良い．なぜなら，任意の内部節点 $s$ の子集合 $\{t_0, t_1, \cdots, t_{m<\sigma}\}$ に対し，$blk(\mathrm{BASE}[s]) = blk(s)$ が定義できれば，自然と $blk(s) = blk(t_0) = blk(t_1) = \cdots = blk(t_m)$ となり，$blk(\mathrm{CHECK}[t_0]) = blk(t_0)$，$blk(\mathrm{CHECK}[t_1]) = blk(t_1)$，$\cdots$，$blk(\mathrm{CHECK}[t_m]) = blk(t_m)$ が満たされるためである．実際には $\sigma = 256$ であるため，$x = 256$ が最も効率が良く，SBASE，SCHECK の記憶量も $\log x = 8$ ビットと小さい．

## 4. 評価

本節はダブル配列，圧縮ダブル配列，LOUDS と比較し評価することで，提案手法の有効性を示す．また，本評価では $x = 256$ とした．

### 4.1 理論的な評価

節 2 で述べた定義において，各データ構造の記憶量に対し理論的な評価を与える．但し，LOUDS に関しては，標準的な実装である tx-trie*4 を参考にした．表 **2** に各手法に対する理論的観測の結果を示す．検索時間における $k$ は入力文字列長であり，$\sigma$ はトライの最大次数を表す．本研究では，文字は 8 ビットで表されることを前提としているため，$\sigma \leq 256$ である．提案手法における $20n$ ビットは，SBASE，SCHECK，BBV，CBV，TERM，LE の記憶量の合計を表しており，$32n(p_b + p_c)$ ビットは LBASE と LCHECK の記憶量の合計を表している．

検索時間については，いずれの手法もノード数に依存せず高速である．しかし，LOUDS に関しては各節点において，入力文字列と一致するラベルを線形探索する必要があり，ダブル配列と比べ低速である．

記憶量について，まず，ダブル配列と提案手法を比較する．表より，ダブル配列の $64n$ から提案手法の $20n$ を差し引いた $44n$ と比べ，$32n(p_b + p_c) < 44n$ すなわち $p_b + p_c < 1.375$ であれば，提案手法の方がダブル配列よりコンパクトになることがわかる．同様に，$32n(p_b + p_c) < 20n$ すなわち $p_b + p_c < 0.625$ であれば，提案手法の方が圧縮ダブル配列よりコンパクトになることがわかる．一方，LOUDS と比べた場合，$20n$ ビットがそもそも LOUDS の記憶量よりも大きいため，LOUDS よりコンパクトになることはない．

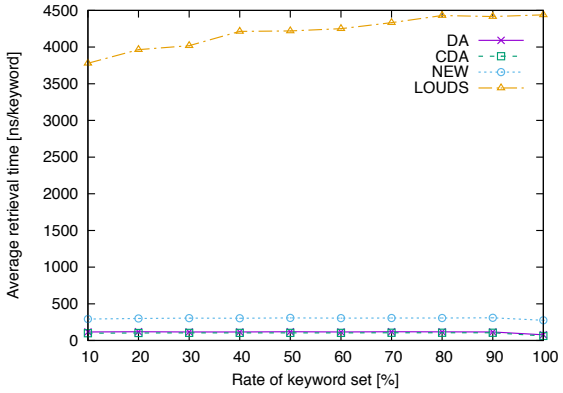


図 4　検索時間に関する実験結果.
**Fig. 4**　Experimental results of retrieval times.

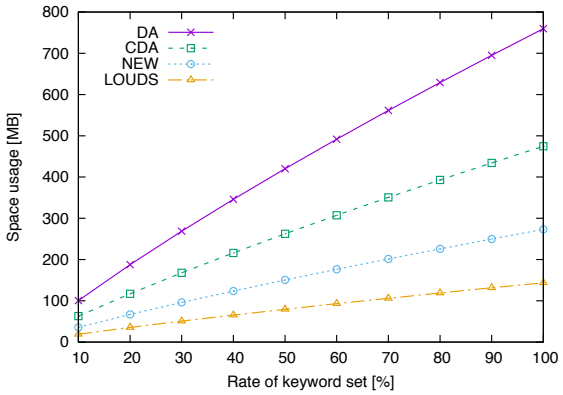


図 5　記憶量に関する実験結果.
**Fig. 5**　Experimental results of space usages.

### 4.2 計算機実験による評価

実環境における提案手法の性能を評価するために，いずれの手法も C++用いて実装し，Quad-Core Intel Xeon 2 x 2.4 GHz(L2 cache: 256 KB) を搭載した計算機上で比較実験をおこなった．LOUDS の実装には tx-trie*4 を用いた．実験では，英語版 Wikipedia の見出し語 *2（キー数：11,519,354，平均長：19.68）によって構築されたトライを表現したときの記憶量と，1 キー当たりの検索時間を求めた．

検索時間に関する実験結果を図 **4**，記憶量に関する実験結果を図 **5** に示す．実験は上記したキー集合の部分集合に対してもおこない．図の横軸は全体集合に対する部分集合の割合を表している．提案手法は，節 3.3 で述べた手順によって構築している．

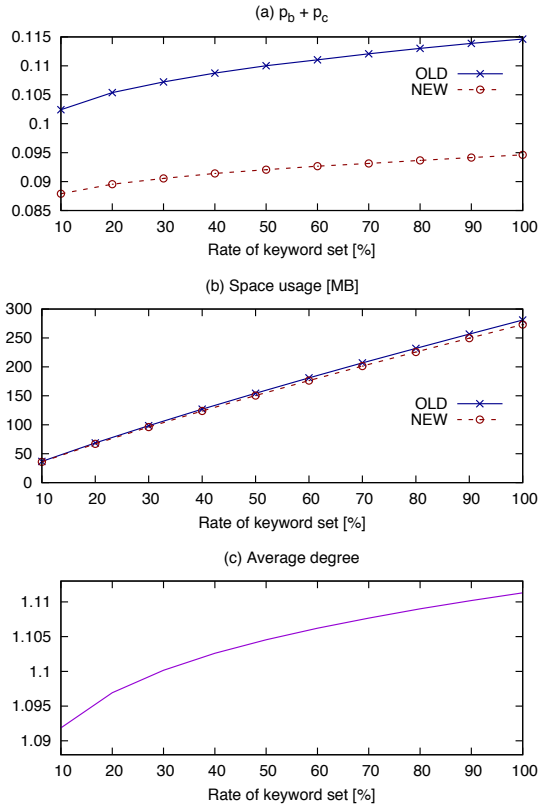


図 6　構築手順と平均次数に対する $p_b$ と $p_c$ 及び記憶量の変化.
**Fig. 6**　Variations of $p_b$, $p_c$ and space usage for construction methods and average degrees.

検索時間については，ダブル配列と圧縮ダブル配列が同等で最も高速であった．提案手法については，理論上は同等だが，遷移における演算が増加したため，ダブル配列と比べて 3 倍ほど低速となった．一方，提案手法と LOUDS を比べた場合，提案手法はダブル配列の特徴を継承しているため，全体集合において約 16 倍高速であった．また，トライの内部節点における平均次数が高くなるほど LOUDS は探索すべき節点が増えるため，全体集合に近づくにつれて低速化が見られたが，タブル配列及び提案手法は次数の影響を受けないため，一定の速度を保ち続けた．記憶量については，提案手法はダブル配列の約 36%，圧縮ダブル配列の約 58%でトライを表現できることがわかった．但し，LOUDS が最も小さく，提案手法の約 53%となった．

加えて，従来の手順と節 3.3 で紹介した手順で構築した場合の $p_b$ と $p_c$ の変化についての実験結果を図 **6** に示す．(a) が，従来の手順（OLD）と節 3.3 で紹介した手順（NEW）で構築した場合の $p_b + p_c$ を示し，(b) がそのときの記憶量を示している．(c) は，トライにおける内部節点の平均次数を示している．横軸は図 4，5 と同じく部分集合の割合を表している．

図より，同じブロックを優先して節点を定義することで $p_b$ と $p_c$ が減少し，記憶量が少し減少したことがわかる．但し，従来の構築手順でも，節 4.1 で述べた圧縮ダブル配列よりコンパクトになる条件 $p_b + p_c < 0.625$ を充分に満たしている．また，キー数とともに平均次数が増加するほど $p_b$ と $p_c$ が増加することがわかる．これは，次数が増加するほど，自身のブロックの空要素を利用しにくくなるためであり，提案手法はブロック長に対し平均次数が小さいほど有効であることがわかった．

以上の結果に加え，逆方向遷移が可能であり動的更新などの機能を持ち合わせていることから，提案手法はトライの表現において有効であるといえる．

## 5. おわりに

本論文では，ダブル配列における従来の圧縮表現の問題点に触れ，逆方向遷移可能かつコンパクトな新しい配列表現を提案した．また，提案手法の記憶効率を更に高める構築手順についても紹介した．そして，実験により，従来のダブル配列よりもコンパクトであり，LOUDS よりも高速であることを示した．

今後の課題としては，様々なキー集合を用いて実験をおこない，$p_b$ と $p_c$ の変化を確認することが挙げられる．また，動的更新を適用した場合に，$p_b$ と $p_c$ が増加しないかなどについても検証する予定である．

## 参考文献

[1] Fredkin, E.: Trie Memory, *Commun. ACM*, Vol. 3, No. 9, pp. 490–499 (1960).
[2] Aoe, J.: An efficient digital search algorithm by using a double-array structure, *IEEE Transactions on Software Engineering*, Vol. 15, No. 9, pp. 1066–1077 (1989).
[3] 矢田　晋，森田和宏，泓田正雄，平石　亘，青江順一：ダブル配列におけるキャッシュの効率化，第 5 回情報科学技術フォーラム講演論文集 (FIT2006)，pp. 71–72 (2006).
[4] 森田和宏，望月久稔，山川善弘，青江順一：トライ構造を用いた共起情報の効率的検索アルゴリズム，情報処理学会論文誌，Vol. 39, No. 9, pp. 2563–2571 (1998).
[5] Fuketa, M., Kitagawa, H., Ogawa, T., Morita, K. and Aoe, J.: Compression of double array structures for fixed length keywords, *Information Processing & Management*, Vol. 50, No. 5, pp. 796–806 (2014).
[6] 神田峻介，泓田正雄，森田和宏，青江順一：階層構造を用いたダブル配列の圧縮法，情報処理学会全国大会講演論文集，No. 1, pp. 693–694 (2015).
[7] Delpratt, O., Rahman, N. and Raman, R.: Engineering the LOUDS succinct tree representation, *Proceedings of WEA 2006*, pp. 134–145 (2006).
[8] Sadakane, K. and Navarro, G.: Fully-functional succinct trees, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, pp. 134–149 (2010).
[9] Munro, J. I. and Raman, V.: Succinct Representation of Balanced Parentheses and Static Trees, *Siam Journal on Computing*, Vol. 31, pp. 762–776 (2001).
[10] Arroyuelo, D., Cánovas, R., Navarro, G. and Sadakane, K.: Succinct Trees in Practice, *Algorithm Engineering and Experimentation*, pp. 84–97 (2010).