Vol.2014-DBS-159 No.11
Vol.2014-IFAT-115 No.11
2014/8/1

近似直線を用いたダブル配列の圧縮法

神田 峻介[1]　森田 和宏[1]　泓田 正雄[1]　青江 順一[1]

概要：トライ法とはキー検索を実現する手法のひとつであり，自然言語処理などにおいて幅広く活用されている．トライ法を実現するデータ構造としては，ダブル配列や LOUDS などがあげられる．ダブル配列は，トライのノード間の遷移を $O(1)$ で実現する高速性を備えたデータ構造であるが，簡潔データ構造である LOUDS と比べ，記憶量は大きい．LOUDS は，ビットベクトルによりトライを表現するため，コンパクト性に優れたデータ構造であるが，ダブル配列に対し検索速度は劣る．本稿では，近似直線との差分値を用いたダブル配列の圧縮法を提案する．また，Wikipedia 日英タイトル各 20 万語〜100 万語に対する実験により，提案手法は従来のダブル配列と比べて，記憶量を約 60%に圧縮し，且つ LOUDS より約 12 倍高速に検索がおこなえることが実証された．

A Compression Method of Double Array Structures using Approximate Straight Lines

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

Abstract: A trie is one of the method for key search algorithm and utilized in natural language processing and so on. It is represented by a double array and LOUDS. The double array provides fast retrieval at time complexty of $O(1)$, but its space usage is larger than that of LOUDS. LOUDS is a succinct data structure using bit-vector. Its space usage is extremely compact, but its retrieval speed is not so fast. This paper presents a compression method of the double array using approximate straight lines. From simulation results for 200,000〜1,000,000 keys, it turned out that the space usage of the presented method becomes about 60% compared with the double array and its retrieval speed is about twelve times faster than that of LOUDS.

1. はじめに

トライ法 [1] とは，自然言語処理などの様々な場面において効率的にキー検索を実現する手法 [2] である．トライは，登録キー集合の各キーの共通接頭辞を併合して作られる木構造であり，検索キーの 1 文字 1 文字を比較することにより検索をおこなうことができる．そのため，キーの長さのみに依存した高速な検索をおこなうことができ，また，探索失敗位置の特定や前方部分文字列の検出が容易である．高速性が高く，且つ高機能であるトライは，スペルチェッカ [3] やコンパイラ [4]，形態素解析器 [5] など，幅広い分野で用いられている．

トライを実現する手法として，検索の高速性と記憶量のコンパクト性を併せ持つダブル配列法 [6] がある．ダブル配列は，$BASE$ と $CHECK$ という 2 つの 1 次元配列により，トライのノードの遷移を実現するデータ構造である．ダブル配列の欠点は，配列の要素に要する記憶領域が大きいことである．$BASE$ には遷移先ノードを示す状態番号と遷移文字の内部表現値との差が格納されており，$CHECK$ には遷移元ノードを示す状態番号が格納されている．$BASE$ と $CHECK$ には，通常，1 つの状態に対し 4 bytes ずつの記憶領域を割り当てる．そのため，ダブル配列の要素数を $elems$ とした場合，ダブル配列の記憶量は $8 \times elems$ bytes となる．すなわち，ダブル配列の各要素の記憶領域を削減することで，ダブル配列の記憶効率は向上する．

現在までに，圧縮ダブル配列 [7] や SAMC[8] などの様々なダブル配列の記憶量を圧縮する手法が提案されている．圧縮ダブル配列は，遷移元の特定を遷移文字によりおこな

[1] 徳島大学
University of Tokushima

う．そのため，$CHECK$ に割り当てる記憶領域は 1 byte となり，$5 \times elems$ bytes の記憶量でトライを実現することができる．SAMC（Single Array with Multi Code）は，トライの深さごとに遷移文字の内部表現値を定義することにより，$BASE$ を削除し，$elems$ bytes の記憶量でトライを実現することができる．しかし，登録キー集合によっては未使用要素が増大し，$elems$ が増加する問題点がある．

また，ダブル配列よりも高い記憶効率でトライを実現する手法として LOUDS（Level-Order Unary Degree Sequence）[9] がある．LOUDS はビットベクトルによりトライを表現するため，記憶効率は非常に高いが，ビット演算を用いるためダブル配列と比べ検索速度は低速となる．

本稿では，$BASE$ の近似直線との差分値を用いることにより，$BASE$ に必要な記憶領域を 2 bytes に削減する手法を提案する．

2. ダブル配列

2.1 概要

青江によって提案されたダブル配列は，$BASE$ と $CHECK$ という 2 つの 1 次元配列を用いてトライのノードの遷移を実現する．トライのノードの状態番号は，ダブル配列の要素のインデックスと対応する．以降，状態番号 s はダブル配列の s 番目の要素を表す．トライにおいて，状態番号 s から状態番号 t へ遷移文字 c による遷移が定義されている場合，ダブル配列は次式を満足する．

$$\begin{aligned} t &= BASE[s] + CODE[c] \\ CHECK[t] &= s \end{aligned} \tag{1}$$

すなわち，状態番号 t は $BASE[s]$ と遷移文字 c の内部表現値 $CODE[c]$ の和で計算され，$CHECK[t]$ には状態番号 s からの遷移であることを定義する s を格納している．$BASE$ と $CHECK$ は，状態番号を表現するため，通常の場合，それぞれ 4 bytes を要する．すなわち，ダブル配列は各要素に 8 bytes の記憶領域を要し，ダブル配列の記憶量は $8 \times elems$ bytes となる．

2.2 圧縮ダブル配列

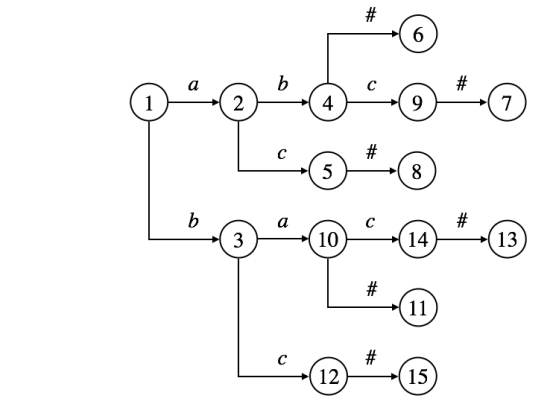
ダブル配列の記憶量圧縮を目的とした研究として，矢田らによって提案された圧縮ダブル配列がある．圧縮ダブル配列は，葉ノードを除くあらゆる状態番号の組 $\{s, u\}$ において，

$$BASE[s] \neq BASE[u] \tag{2}$$

が成立するとき，次式で遷移を実現する．

$$\begin{aligned} t &= BASE[s] + CODE[c] \\ CHECK[t] &= c \end{aligned} \tag{3}$$

圧縮ダブル配列では，遷移元の状態番号の代わりとして，遷移文字を $CHECK$ に格納することにより，$CHECK$ に



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
BASE	1	2	9	6	8				7	11		15		13	
CHECK		*a*	*b*	*b*	*c*	#	#	#	*c*	*a*	#	*c*	#	*c*	#

	#	*a*	*b*	*c*
CODE	0	1	2	3

図 1　圧縮ダブル配列

割り当てる記憶領域を 1 byte で実現している．すなわち，圧縮ダブル配列は各要素に 5 bytes の記憶領域を要し，圧縮ダブル配列の記憶量は $5 \times elems$ bytes となる．

キー集合 $K = \{$"ab","abc","ac","ba","bac","bc"$\}$ に対する圧縮ダブル配列を図 1 に示す．ここで，$'\#'$ は文字列の終端を表している．

3. 提案手法

3.1 概要

提案手法では，圧縮ダブル配列を構築する際，本手法により定義した近似直線との差分値が，2 bytes で表現可能な値として $BASE$ 値を決定する．このとき，同時に式 2 も満たす必要があるため，従来の構築方法では決定において衝突が生じやすく，且つ最適な近似直線の定義は困難である．そのため，提案手法では泓田らの手法 [8] と同様にトライを深さごとに分割し，深さごとに近似直線 $f_d(s)$ を定義する．ここで $d(\geq 1)$ はトライにおける深さ，s は状態番号を表す．このとき，提案手法では，次式によりトライの遷移を実現する．

$$\begin{aligned} t &= DBASE[s] + f_d(s) + CODE[c] \\ CHECK[t] &= c \end{aligned} \tag{4}$$

ここで，$DBASE$ は各要素に割り当てる記憶領域が 2bytes の 1 次元配列である．また，近似直線 $f_d(s)$ は状態番号を実変数とする線形関数であり，次式で表す．

$$f_d(s) = \mathrm{a}_d s + \mathrm{b}_d \tag{5}$$

以降，$f_d(s)$ は小数点以下を切り捨て，整数値として計算する．また，圧縮ダブル配列の遷移を表す式 3 と式 4 は同義であり，次式が成立する．

Vol.2014-DBS-159 No.11
Vol.2014-IFAT-115 No.11
2014/8/1

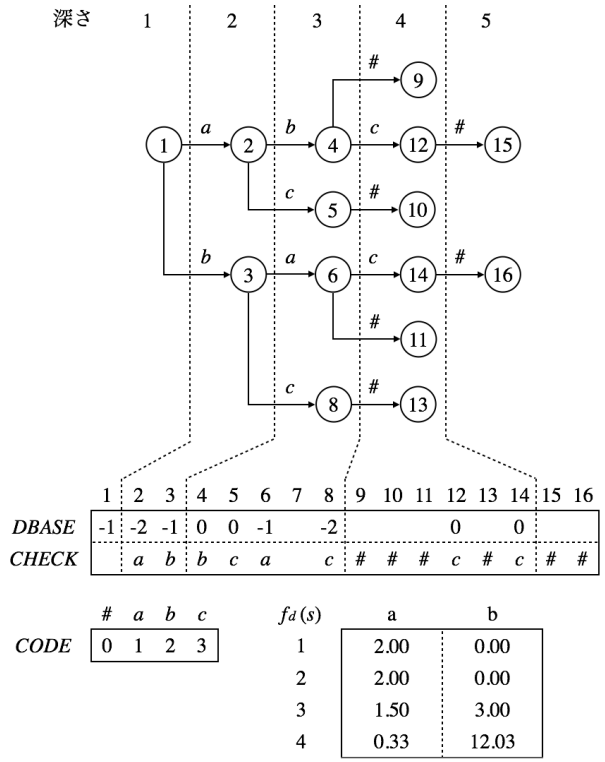


図 2　提案手法

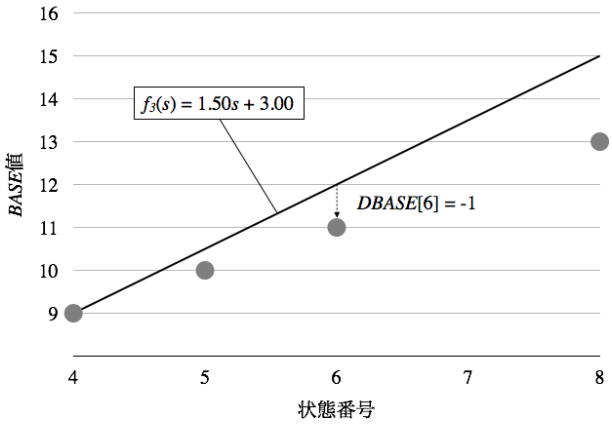


図 3　状態番号-$BASE$ 値の散布図，及び線形関数 $f_3(s)$

$$BASE[s] = DBASE[s] + f_d(s) \tag{6}$$

ここで，式 2 と同様に，葉ノードを除くあらゆる状態番号の組 $\{s, u\}$ において，次式が成立する必要がある．

$$DBASE[s] + f_d(s) \neq DBASE[u] + f_d(u) \tag{7}$$

提案手法では，この $f_d(s)$ と配列 $DBASE$ を用いてダブル配列を実現する．そのため，提案手法によるダブル配列の記憶量は $3 \times elems$ bytes となる．

キー集合 K に対する提案手法によるダブル配列を図 2 に示す．このとき $DBASE$ は，式 6 を満たす値であり，$f_d(s)$ との差分値であるため，図 1 よりも小さな値で実現できる．また，図 2 の深さ 3 における状態番号と $BASE$ 値の散布図と，$f_3(s)$ を図 3 に示す．

3.2　構築概要

提案手法では，$DBASE$ の各要素の記憶領域を 2 bytes で実現する．このとき，以下の条件を満たす必要がある．

条件 1　あらゆる状態番号 s に対し，$f_d(s)$ との差が 2 bytes で表現可能な値として，$BASE[s]$ を決定する．　□

本節では，条件 1 を満たす構築手法について述べる．

3.2.1　線形関数 $f_d(s)$

提案手法では，トライの最深部の深さが n であったとき，n 個の線形関数をトライの深さごとに定義する．このとき，ダブル配列の要素は，深さごとに互いに独立した範囲を持つ．以下にその範囲を示す．

$$s\_min_d = \begin{cases} 1 & (d = 1) \\ s\_max_{d-1} + 1 & (d \geq 2) \end{cases} \tag{8}$$

$$s\_max_d = \sum_{k=1}^{d} (nodes_k + unused_k) \tag{9}$$

このとき，s_min_d は深さ d に対応した状態番号の最小値であり，s_max_d は深さ d に対応した状態番号の最大値である．また，$nodes_d$ は深さ d におけるノードに対応した要素数，$unused_d$ は深さ d に対応する要素の中で未使用状態である要素数である．例えば，図 2 のダブル配列では，状態番号 7 が提案手法により発生した未使用状態の要素であり，$nodes_3 = 4$，$unused_3 = 1$ である．

深さ d の $BASE$ 値を決定するとき，線形関数 $f_d(s)$ に応じて，条件 1 を満たす値として決定する．このとき，$BASE$ 値の決定には，従来のダブル配列 [6] と同様に，前方の未使用要素から優先的に利用する．ここで，提案手法では，$f_d(s)$ を単調増加関数として定義し，ダブル配列における状態番号順に $BASE$ 値を決定する．また，$f_d(s)$ における a_d が大きくなるほど，深さ d における $BASE$ 値の取り得る範囲は大きくなる．

$f_d(s)$ における a_d は，深さ d におけるダブル配列の要素の範囲と，取り得る $BASE$ 値の範囲を用いて定義する．ここで，遷移文字の内部表現値を状態番号と比べ無視できるほど小さい値と見なすと，$BASE[s\_min_d, s\_max_d]$[*1] が取り得る値の範囲は，$[s\_min_{d+1}, s\_max_{d+1}]$ と見なすことができる．そのため，$f_d(s)$ は，状態番号と $BASE$ 値の線形空間上の座標 (s_min_d, s_min_{d+1}) と (s_max_d, s_max_{d+1}) を通る直線として定義する．このとき，式 8，9 より，$[s\_min_d, s\_max_d]$ の要素数は，$nodes_d + unused_d$ である．また，$[s\_min_{d+1}, s\_max_{d+1}]$ の要素数は，$nodes_{d+1} + unused_{d+1}$ である．これより，a_d は次式とできる．

$$\mathrm{a}_d = \frac{nodes_{d+1} + unused_{d+1}}{nodes_d + unused_d}$$

しかし，$unused_{d+1}$ は深さ d の要素に対する $BASE$ の決

[*1] 本稿では，変数の i から，$i \leq j$ までの区間を $[i, j]$ により表す．また，配列 A の i 番目の要素から，$i \leq j$ までの要素列を $A[i, j]$ により表す．

定段階では未定である．そのため，未使用要素 $unused_{d+1}$ は無視できるほど充分に小さい値と見なし，次式により傾き a_d を定義する．

$$\mathrm{a}_d = \frac{nodes_{d+1}}{nodes_d + unused_d} \tag{10}$$

また，式 5 が座標 (s_min_d, s_min_{d+1}) を通ることより，b_d は次式となる．

$$\mathrm{b}_d = s\_min_{d+1} - \mathrm{a}_d s\_min_d$$

しかし，s_min_{d+1} は深さ d の要素に対する $BASE$ の決定段階では未定である．そのため，式 8 において

$$s\_min_{d+1} = s\_max_d + 1$$

であることに着目し，次式により b_d を定義する．

$$\mathrm{b}_d = s\_max_d + 1 - \mathrm{a}_d s\_min_d \tag{11}$$

また，登録キー集合によっては，式 10 では $BASE$ 値を $f_d(s)$ との差分値が 2 bytes で表現可能な値として決定できない場合がある．その場合，傾き a_d を加算し，$BASE[s\_min_d, s\_max_d]$ が取り得る値の範囲を拡大させる必要がある．そして，深さ d から深さ $d+1$ への遷移における $BASE$ 値を再度決定し直し，再構築をおこなう．このとき，a_d の加算は次式によりおこなう．

$$\mathrm{a}_d = \frac{nodes_{d+1}}{nodes_d + unused_d} + gain_d \tag{12}$$

ここで，$gain_d$ は a_d に対する加算値であり，再構築をおこなう上で，条件 1 を実現する最適な値として設定する必要がある．しかし，最適な $gain_d$ を設定するためには，$gain_d$ に様々な値を設定し，試行を繰り返す必要がある．

3.2.2 *BASE* 値の収束範囲

提案手法では，式 5，式 10，式 11 で定義される $f_d(s)$ と $BASE[s]$ の差分値を $DBASE[s]$ に格納する．以降では，条件 1 を満たし得る，$f_d(s)$ に対する $BASE$ 値の収束範囲について，集合 M_s と集合 L_s を用いて述べる．また，条件 1 を満たし得る $BASE$ 値の収束範囲は，定数 α を用いて調整する．集合 M_s は，条件 1 を満たし得る $BASE$ 値において，$f_d(s)$ より大きな値の集合であり，以下を満たす整数値の集合である．

$$M_s = \{x \mid f_d(s) < x \leq f_d(s) + 2^{16} - \alpha\} \tag{13}$$

集合 L_s は，条件 1 を満たし得る $BASE$ 値において，$f_d(s)$ 以下の値の集合である．L_s は，以下の条件において，次式を満たす整数値の集合である．

(i) $s\_min_d \leq s < s'$

$$L_s = \{x \mid s\_max_d + 1 - code \leq x \leq f_d(s)\} \tag{14}$$

(ii) $s' \leq s \leq s\_max_d$

$$L_s = \{x \mid f_d(s) - \alpha \leq x \leq f_d(s)\} \tag{15}$$

ここで，$code$ は，遷移文字の内部表現値の取り得る最大値を表す．また，s' は，$f_d(s) - \alpha$ と $BASE[s\_min_d]$ が交わる s の値であり，s' は以下の式で求めることができる．

$$s' = \frac{s\_max_d + 1 - \mathrm{b}_d + \alpha}{\mathrm{a}_d} \tag{16}$$

このとき，条件 1 を満たす $BASE$ 値は次式を満たす値である．

$$BASE[s] \in \{M_s \cup L_s\} \tag{17}$$

以降で，式 13，式 15，式 16 で用いた，$BASE$ 値が取り得る集合を制限するための定数 α について述べる．条件 (i) を満たす s の場合，定数 α の値が大きくなりすぎると，式 13 より，$n(M_s)$[*2] が少なくなる．$n(M_s)$ が少なくなるに伴い，式 14 より，$BASE$ 値の取り得る要素の集合は，条件 (ii) の時と比べ少なくなる．そのため，M_s の要素の集合から $BASE$ 値を取得できない場合があり，提案手法を実現できない．また，条件 (ii) を満たす s の場合，定数 α の値が小さすぎると，式 15 より，$n(L_s)$ が減少し，$BASE$ 値に用いられる未使用要素の集合のうち，$f_d(s)$ 以下の使用不可能な要素が増大する．そのため，ダブル配列全体の未使用要素が増大し，記憶量が大きくなってしまう．以上より，定数 α には適切な値を設定する必要がある．

3.3 構築アルゴリズム

提案手法によるダブル配列の構築アルゴリズムを Algorithm 1 に示す．

手続き **build** は，別のデータ構造を用いて既に構築したトライを構築土台とし，提案手法によるダブル配列を構築する．ここで，手続き **build** では，構築土台となるトライに対し，**label** と **childs** という二つの関数を用いる．**label** は，状態への遷移文字を返す関数であり，**childs** は，状態の遷移先ノードを配列として返す関数である．また，構築土台となるトライの状態番号と，ダブル配列における状態番号の対応を確認するために，配列 N を用いる．例えば，構築土台となるトライの状態番号 s には，ダブル配列の状態番号 $N[s]$ が対応する．$state\_set$ は，構築土台となるトライを幅優先順に遷移するために用いられるコンテナである．このコンテナは，構築土台となるトライの状態番号を要素とし，各深さに対応したコンテナを用意する．また，$root\_state, root\_depth$ は，構築土台となるトライの根ノードに対応する状態番号と深さであり，max_depth は，構築土台となるトライの最も深いノードに対する深さである．第 1 行では，**append** により根ノードの深さに対応した $state\_set$ へ根ノードの状態番号を追加している．

第 3 行から開始するループでは，構築土台となるトライ

[*2] $n(A)$ は有限集合 A の要素の個数を表す．

を深さごとに辿り，提案手法によるダブル配列を構築する．ここで，第 4 行の **set_linear** は，線形関数 $f_d(s)$ を式 10, 11 により設定する関数である．また，第 5 行の **sort** は，$state\_set$ をダブル配列の状態番号順で整列させる関数である．

第 6 行から開始するループでは，深さ d におけるダブル配列を構築する．ここで，**build_depth** とは，深さ d のノードに対しダブル配列を構築する関数であり，再構築が必要な場合は False を返す．このとき，第 7 行では，**init_da** により，$DBASE[s\_min_d, s\_max_d]$, $CHECK[s\_min_{d+1}, s\_max_{d+1}]$ と $state\_set_{d+1}$ を初期化する．また，第 8 行では，**update_linear** で，式 12 により線形関数を更新する．このとき，**update_linear** は，第 6 行のループにより，構築が完了するまで複数回呼び出され，2 回目以降の呼び出しにおいては，前回の $gain\_d$ より大きな値を設定する．

次に，第 6 行における **build_depth** のアルゴリズムを Algorithm 2 に示す．関数 **build_depth** は，引数として受け取った深さについてダブル配列を構築し，構築に成功したか否かを真偽値で返す．

第 1 行から開始するループでは，深さ d におけるノードを辿り，ダブル配列を構築する．ここで，第 2 行の **xcheck** とは，式 2 を満たし，且つ $L_{N[s]}$ の要素の最小値以上の値のうち，小さい値を優先的に返す関数である．第 3 行では，$BASE$ 値が式 17 を満たす値であるかを判断する．もし，真であった場合は，第 4 行で，$BASE$ 値と線形関数の差分値を $DBASE$ 値に設定する．偽の場合は，$BASE$ 値が式 17 を満たさない値，すなわち，$M_{N[s]}$ の要素の最大値より大きい値であり，再構築をおこなう必要があるため False を返す．

第 8 行から開始するループでは，s に対する子ノードを全て辿り，ダブル配列の状態番号と $CHECK$ 値の設定，及び次の深さの構築に向けての設定をおこなう．ここで，第 11 行では，t が葉ノードであるかを調べ，葉ノードでない場合は次の深さに対応する $state\_set$ に子ノードの状態番号を追加する．

以上のアルゴリズムにより，ダブル配列を構築する．

4. 評価

提案手法の有効性を実証するために比較実験をおこなった．プロセッサは 2 x 2.4 GHz Quad-Core Intel Xeon によりおこなった．また，登録キー集合として，英語版 Wikipedia タイトル [*3]，日本語版 Wikipedia タイトル [*4] から，それぞれ無作為に各 20 万語〜100 万語を取得し，使用した．登録キー集合の文字コードは UTF-8 とした．比較対象は，圧縮ダブル配列と LOUDS とし，LOUDS の検証

[*3] http://dumps.wikimedia.org/enwiki/20131104/
[*4] http://dumps.wikimedia.org/jawiki/20131107/

Algorithm 1 Procedure **build**

Require: *a base trie structure*
Ensure: $state\_set_{root\_depth} = ... = state\_set_{max\_depth} := \phi$
1: **append**($state\_set_{root\_depth}, root\_state$)
2: $N[root\_state] := root\_state$
3: **for** $d := root\_depth$ **to** max_depth **do**
4: **set_linear**(d)
5: **sort**($state\_set_d$)
6: **while build_depth**(d) = False **do**
7: **init_da**(d)
8: **update_linear**(d)
9: **end while**
10: **end for**

Algorithm 2 Function **build_depth**

Require: d represents the depth of the base trie structure.
1: **for** s **in** $state\_set_d$ **do**
2: $base :=$ **xcheck**(s)
3: **if** $base \in \{M_{N[s]} \cup L_{N[s]}\}$ **then**
4: $DBASE[N[s]] := base - f_d(N[s])$
5: **else**
6: **return** False
7: **end if**
8: **for** t **in childs**(s) **do**
9: $N[t] := base + CODE[$**label**$(t)]$
10: $CHECK[N[t]] :=$ **label**(t)
11: **if childs**(t) $\neq \phi$ **then**
12: **append**($state\_set_{d+1}, t$)
13: **end if**
14: **end for**
15: **end for**
16: **return** True

表 1 ダブル配列の要素数 $elems$（千個）

登録キー数	20 万	40 万	60 万	80 万	100 万
英語					
圧縮ダブル配列	2,511	4,780	6,972	9,097	11,152
提案手法	2,548	4,858	7,108	9,305	11,449
日本語					
圧縮ダブル配列	2,663	5,013	7,234	9,352	11,429
提案手法	2,701	5,073	7,308	9,445	11,538

にはライブラリ tx-trie-0.18[*5] を用いた．検索速度は，1 つのキーに対する平均検索時間を計測した．遷移文字の内部表現値は，キー集合における遷移文字の出現頻度順に，0 から割り当てた．また，提案手法における定数 α は 12,000 とし，$gain_d$ は再構築の度に 0.03 ずつ加算した．

記憶効率に関する結果を表 1，図 4，5 に示す．実験結果より，圧縮ダブル配列と比べ，提案手法は多少の要素数の増加は見られるものの，記憶量を約 60%に圧縮できることがわかった．また，検索速度に関する結果を図 6，7 に示す．実験結果より，LOUDS と比べ，提案手法は約 12 倍

[*5] https://code.google.com/p/tx-trie/

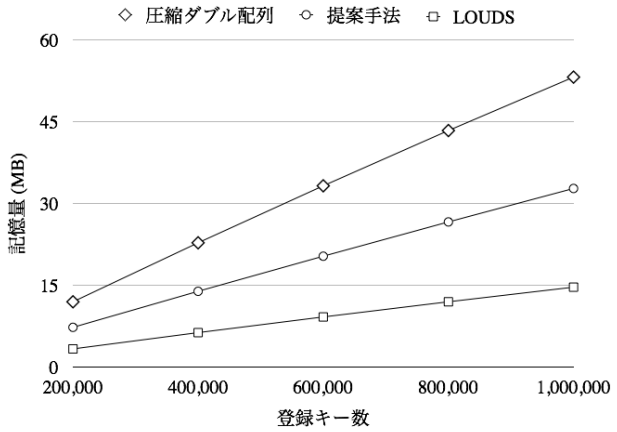


図 4　記憶効率に対する実験結果（英語）

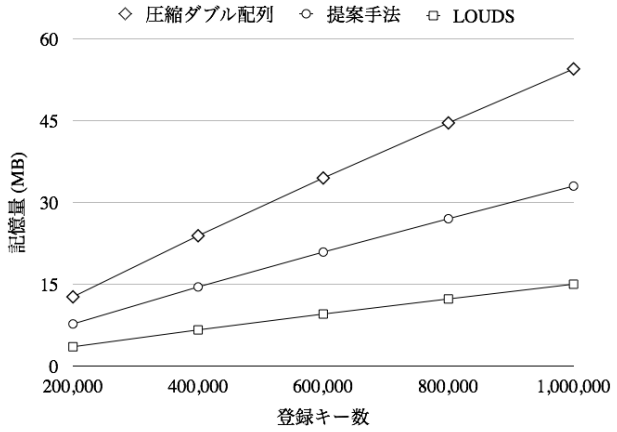


図 5　記憶効率に対する実験結果（日本語）

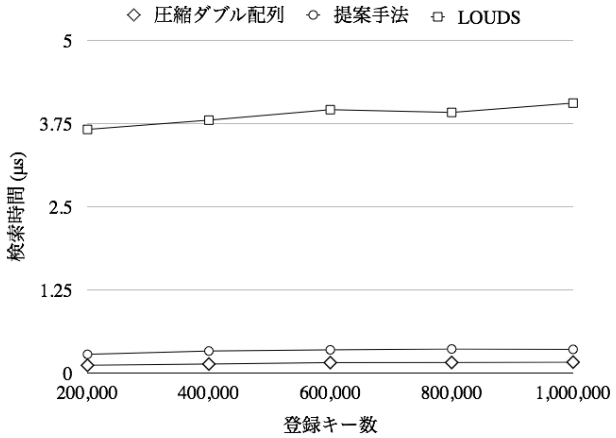


図 6　検索速度に対する実験結果（英語）

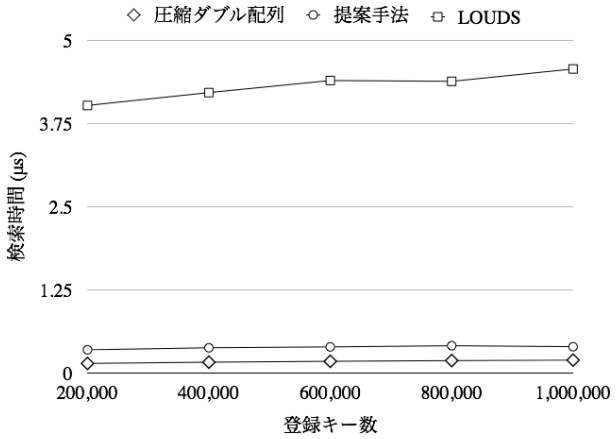


図 7　検索速度に対する実験結果（日本語）

高速に検索をおこなえることがわかった．また，再構築は英語版 Wikipedia タイトルの 100 万語に対して一度のみであった．

5. おわりに

本稿では，キー検索法の一つであるダブル配列法について，$BASE$ 値の近似直線を用いることにより，ダブル配列の各要素に必要な記憶領域を削減し，ダブル配列の記憶量を圧縮する手法を提案した．そして，比較実験により提案手法の有効性を示した．

提案手法の今後の課題は，最適な α，$gain_d$ を求める定義を明確にし，キー集合の規模にかかわらず提案手法が実現可能であることを理論的に証明することが挙げられる．

参考文献

[1] 青江順一：トライとその応用 (< 連載講座 > キー検索技法 4)，情報処理， Vol. 34, No. 2, pp. 244–251 (1993).
[2] Aoe, J.: *Computer Algorithms: Key Search Strategies*, IEEE Computer Society Press, Los Alamitos, CA, USA (1991).
[3] Peterson, J.: *Computer Programs for Spelling Correction*, Lecture Notes in Computer Science, Springer-Verlag (1980).
[4] Aho, A. V. and Ullman, J. D.: *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1977).
[5] 田中穂積：自然言語解析の基礎，産業図書 (1989).
[6] Aoe, J.: An efficient digital search algorithm by using a double-array structure, *Software Engineering, IEEE Transactions on*, Vol. 15, No. 9, pp. 1066–1077 (1989).
[7] Yata, S., Oono, M., Morita, K., Fuketa, M., Sumitomo, T. and Aoe, J.: A compact static double-array keeping character codes, *Information Processing & Management*, Vol. 43, No. 1, pp. 237–247 (2007).
[8] Fuketa, M., Kitagawa, H., Ogawa, T., Morita, K. and Aoe, J.: Compression of double array structures for fixed length keywords, *Information Processing & Management* (in press).
[9] Delpratt, O., Rahman, N. and Raman, R.: Engineering the LOUDS Succinct Tree Representation, *WEA 2006*, pp. 134–145 (2006).