ダブル配列構築の高速化を目的とした 節点から遷移可能な遷移種の集合に基づく未使用要素の管理法

村山　智也†　　望月　久稔††

† 大阪教育大学大学院教育学研究科総合基礎科学専攻　〒 582–8582 大阪府柏原市旭ヶ丘 4-698-1
†† 大阪教育大学教育学部教養学科情報科学講座　〒 582–8582 大阪府柏原市旭ヶ丘 4-698-1
E-mail: †j149610@ex.osaka-kyoiku.ac.jp, ††motizuki@cc.osaka-kyoiku.ac.jp

あらまし　文書の電子化などにより，我々が扱う情報量は増加を続けている．それら大量の情報を効率的に扱うには高速な検索手法が必要となる．文書を対象とする言語処理における高速な辞書検索手法にトライ木があり，検索性能を維持したまま領域を削減する実装法としてダブル配列がある．ダブル配列は，トライ木において各節点から遷移可能な遷移種の集合を一次元配列上で組み合わせた構造をとる．そのため，構築の際に遷移種の集合が当てはまる位置を計算する必要があり，時間計算量を要する．また，増加し続ける情報をリアルタイムに扱うためには，より高速な構築が求められる．そこで，本研究はダブル配列構築の高速化を目的として，一次元配列における周囲の使用状況に応じて未使用要素を分類することで，遷移種の集合が当てはまる位置の算出に必要な比較回数の削減を図る．実験の結果，比較回数を 60〜95 %削減し，構築時間を 7〜31 %短縮した．

キーワード　キー検索技法，木構造，ダブル配列

1. はじめに

電子コンテンツの拡充や文書の電子化などにより，我々が扱う情報量は増加を続けている．それら大量の情報を効率的に扱うには高速な検索手法が必要となる．文書を対象とする言語処理において，高速な辞書検索手法にトライ木があり，これを実現するデータ構造としてダブル配列がある [1] [2]．ダブル配列は，トライ木において節点から遷移可能な遷移種の集合，すなわち遷移集合を 2 本の 1 次元配列上で組み合わせた構造をとり，トライ木の高速な検索性能を引き継ぎつつ領域を圧縮する．コンパクトかつ高速な検索性能から，形態素解析エンジン MeCab [3] や ChaSen [4] の辞書として利用されている．また，ダブル配列を拡張してトライ木以外を実装する研究もなされており，蔵満らはマシン AC によるアンチウイルスエンジン [5] や GLR 構文解析器 [6] を提案した．

今なお増加を続けるリアルタイムな情報を扱うためには，パターン認識システムの高速な構築法が必要となる．しかし，ダブル配列は節点の遷移集合を 1 次元配列上で組み合わせる位置の算出に時間計算量を要する．そこで本研究は，ダブル配列構築の高速化を目的として，ダブル配列における未使用要素を周囲の未使用状況に応じて分類することで，遷移集合が当てはまる位置の算出に必要な比較回数の削減をはかる．

2. ではダブル配列の構築と問題点を説明し，3. では未使用要素を分類して管理する未使用要素リストの構築法と利用法を説明する．4. で提案手法に対して評価を与えた後，5. で総括する．

2. ダブル配列の構築と遷移の衝突

ダブル配列はトライ木における節点の遷移を 2 本の 1 次元配列 BASE と CHECK により定義する．遷移種 a による節点 r から節点 t への遷移が存在するとき，ダブル配列は式 (1),(2) を満たす．基底値の配列 BASE は節点 r の遷移集合を組み合わせる際の基底値 $BASE[r]$ を格納し，節点番号の配列 CHECK は遷移元の節点 $CHECK[r]$ を格納する．ここで，配列の位置が共通する基底値 $bse = BASE[r]$ と遷移元節点番号 $chk = CHECK[r]$ の組をダブル配列における要素とする．トライ木とダブル配列の要素との対応を図 1 に示す．

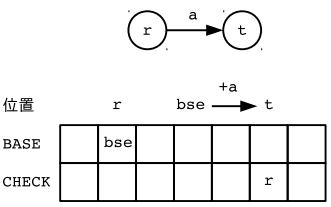

図 1　ダブル配列の遷移

$$BASE[r] + a = t \tag{1}$$

$$CHECK[t] = r \tag{2}$$

以降，トライ木の節点を表す要素を使用要素，それ以外を未使用要素と定義する．ダブル配列における全要素の集合を D，全使用要素の集合を N，全未使用要素の集合を F としたとき，集合 D, N, F の関係式を式 (3) に示す．

$$D = N \cup F,\ N \cap F = \phi \tag{3}$$

ダブル配列の構築法を説明する．逐次的にキーを追加する関数 Insert を図 2 に示す．ここで，以下の関数を定義する．

- **Collision(node, c)** 節点 node から遷移種 c による遷移先要素が使用要素である場合，その使用要素か節点 node の

関数：Insert
引数：追加するキー *key*

(I-1)c ← key における先頭の遷移種
(I-2)for (*node* ← 根; *node* が葉でない; *node* ← *next*)
(I-3)　*next* ← BASE[*node*] + c
(I-4)　if (CHECK[*next*] ≠ *node*)
(I-5)　　break
(I-6)　c ← key における次の遷移種
(I-7)if (*next* ∈ *N*)
(I-8)　node ← Collision(node, c)
(I-9)MakeSingle(node, 遷移に使わなかった key の固有接尾辞)

図 2　関数 Insert

遷移先節点を移動する．使用要素を移動する際，節点 node が移動し得るため，移動後の節点 node を返す．

- **MakeSingle(node, str)** 節点 node から遷移列 str を追加する．

トライ木はキー集合の共通接頭辞を併合した構造であるため，キーを追加する際は，追加済みのキーとの共通接頭辞を遷移した後，追加するキーの固有接尾辞を節点として追加する．図 2 中手順 (I-2) から (I-6) により，追加するキーを遷移列とみなして共通接頭辞を遷移する．(I-7) で遷移先 *next* が使用要素である，つまり遷移が衝突した場合，衝突した要素か衝突された要素を移動する関数 Collision を呼び出す．衝突の解決後，キーの固有接尾辞をダブル配列上に節点として登録する．(注1)．

[例 1] ダブル配列構築における遷移の衝突を説明する．図 3 は，キー集合 $K = \{$"aaa","aae","ae","d"$\}$ を登録したダブル配列に対してキー "b" を追加する際に衝突した状況を表す．ここで，遷移種の内部表現値は $\{$'a' $= 1$, 'b' $= 2$, 'c' $= 3, \cdots\}$ と定義する．図の上部にトライ木と下部にダブル配列を表す．トライ木における二重丸は遷移の終点である葉節点を表し，ダブル配列では *bse* を負にすることで定義する．キー "b" を追加するために根の要素である 1 番節点から遷移種 'b' による遷移先は，式 (1) より $BASE[1] +$ 'b' $= 2 + 2 = 4$ となり，4 番節点となる．しかし，4 番節点はすでに 3 番節点の遷移種 'a' による遷移先として使用されているため，遷移が衝突する．　**[例終]**

衝突を解決するためには，衝突した要素か衝突された要素を移動する必要がある．ダブル配列の定義から，要素を移動する際は同じ親節点をもつ要素も移動する．そのため，親節点がもつ遷移集合が当てはまる *bse* をダブル配列上で算出する必要がある．図 3 の場合，1 番要素から遷移種 'b' による遷移先が衝突しない位置を探すために，1 番要素の遷移集合 $s = \{$'a', 'b', 'd'$\}$ が当てはまる *bse* を算出する．以降，遷移集合がダブル配列上で当てはまる位置を探し，対応する *bse* を返す関数を XCheck とよぶ．青江らの XCheck に要する時間計算量は非常に大きいため [7]，これを高速化する先行研究として，未使用要素のリストを単方向リンクリストで管理する森田らの手法 [7]，双方向リンクリストで管理する大野ら [8]，矢田ら [9]，中村らの手法 [10] などがある．図 4 は図 3 のダブル配列における未使用要素 $\{2, 5, 9, 10, 12, 13, \cdots\}$（図 3 のダブル配列における丸印の無い位置）を 1 本のリンクリストで管理する．

(注1)：固有接尾辞を，TAIL と呼ばれる配列に登録する手法 [1] と，ダブル配列上に節点として登録する手法がある [7] が，本稿では TAIL 配列を使用する．

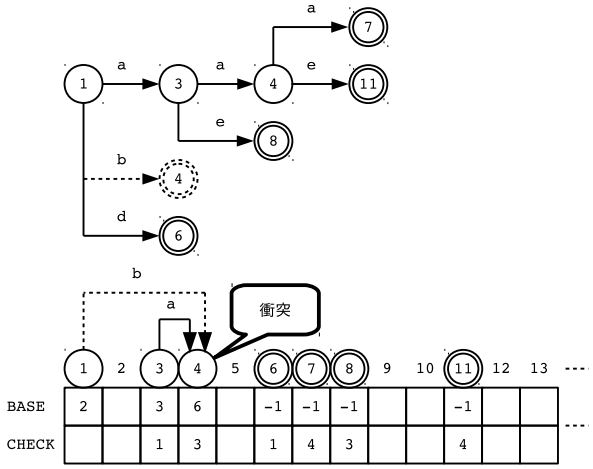

図 3　ダブル配列構築における遷移の衝突

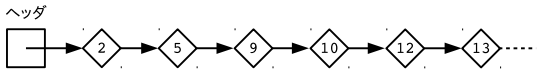

図 4　従来の未使用要素リスト

関数：XCheck
引数：遷移集合 *s*
戻り値：*s* が当てはまる *bse*

(X-1)foreach *node* in *FL*
(X-2)　*bse* ← *node* - *a*; ∃*a* ∈ *s*
(X-3)　if (遷移集合 *s* が *bse* から当てはまる)
(X-4)　　*bse* を返す

図 5　関数 XCheck

未使用要素リストにつながる未使用要素をたどり，遷移集合が当てはまる *bse* を算出する関数 XCheck を図 5 に示す．リンクリストで管理する未使用要素の集合を *FL* と表す．図中手順 (X-1) により，リストにつながる未使用要素を順番にたどる．(X-2) で未使用要素 *node* と遷移集合 *s* に含まれる遷移種 *a* から *bse* を求め，(X-3) で遷移集合 *s* に含まれる全ての遷移が他の遷移と衝突しないか判定する．衝突しない *bse* を見つけた場合，(X-4) で *bse* を返す．

[例 2] 図 5 の XCheck により，図 4 の未使用要素リストから遷移集合 $s = \{$'a', 'b', 'd'$\}$ が当てはまる *bse* を算出する例を説明する．まず，未使用要素 *node* を 2 番要素としたとき，*bse* は $2 - 1(=$ 'a'$) = 1$ である．しかし，遷移種 'b' に対応する遷移先である 3 番要素は使用要素であるため衝突する．次の *node* である 5 番要素も同様に衝突する．次の *node* である 9 番要素は *bse* が 8 であり，遷移種 'a' は未使用要素 *node*，遷移種 'b' は未使用要素 10，遷移種 'd' は未使用要素 12 にそれぞれ対応し，遷移集合 *s* が当てはまる $bse = 8$ を算出する．以上より，未使用要素 2，5，9 を比較して *bse* が求まる．　**[例終]**

関数：FreeSituation
引数：未使用要素 r
戻り値：未使用状況 h

(FS-1)for $(i \leftarrow 0; i < m; i{+}{+})$
(FS-2)　if ($(r+1+i)$ 番要素 $\in F$)
(FS-3)　　h.bit$[i] \leftarrow 1$
(FS-4)　else
(FS-5)　　h.bit$[i] \leftarrow 0$
(FS-6) 未使用状況として h を返す

図 6　関数 FreeSituation

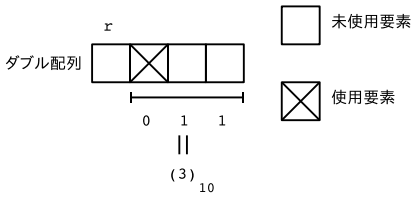

図 7　未使用状況の例

3. 未使用状況に基づく未使用要素管理法

従来手法による bse の算出では，全ての未使用要素が比較候補であるため，遷移集合が当てはまらない未使用要素を比較する可能性がある．そこで提案手法は，未使用要素を周囲の未使用状況に基づいて分類し，複数本の未使用要素リストを構築する．XCheck で bse を算出する際は，遷移集合に基づいて未使用要素リストを選択することで，XCheck における比較回数の削減と構築時間の短縮をはかる．

本章では，隣接未使用状況に基づく未使用要素リストの構築法を説明後，分類した未使用要素リストを用いて未使用要素の比較回数を削減する XCheck を説明する．

3.1　未使用要素リストの構築

周囲の未使用状況によって未使用要素を分類するために，提案手法は未使用状況を数値化する．未使用要素 r の未使用状況を求める関数 FreeSituation を図 6 に示す．FreeSituation は，配列上の未使用要素 r の右に隣接する要素 m 個を図中手順 (FS-1) で参照し，(FS-2)〜(FS-5) で未使用要素を 1，使用要素を 0 と定義する．長さ m のビット列により未使用状況を数値化する．

[例 3] 未使用状況の例を図 7 に示す．ここで $m = 3$ と定義する．図中，未使用要素 r の右に隣接する 3 要素は使用要素，未使用要素，未使用要素と並ぶため，ビット列 011 と表す．$(011)_2 = (3)_{10}$ より，r の未使用状況は 3 となる．　**[例終]**

数値化した未使用状況は $(00\cdots0)_2$〜$(11\cdots1)_2$ の 2^m 通りが考えられる．そこで，未使用要素リストのヘッダとして要素数 2^m の配列を用意し，配列の位置と数値化した未使用状況とを対応させる．未使用状況 h に対応する未使用要素リスト $FL[h]$ は式 (4) を満たす．

$$FL[h] = \{\ r \mid FreeSituation(r) = h\ \}$$
$$h \in \mathbb{Z}, 0 \leqq h < 2^m \qquad (4)$$

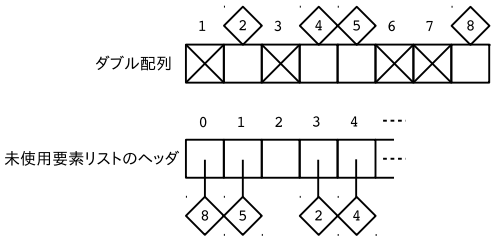

図 8　未使用要素リストの例

関数：XCheck
引数：遷移集合 s
戻り値：s が当てはまる bse

(XP-1)$h \leftarrow$ TransSetToHead(s)
(XP-2)foreach $node$ in $FL[h]$
(XP-3)　$bse \leftarrow node$ - s_{min}
(XP-4)　if (遷移集合 s が bse から当てはまる)
(XP-5)　　bse を返す

図 9　関数 XCheck の変更版

[例 4] 構築の過程における未使用要素リストの例を図 8 に示す．図中，上部の配列はダブル配列の要素を表し，下部は未使用要素リストを表す．このとき，未使用要素 2 の未使用状況は，例 3 より $(011)_2 = 3$ であるので，未使用要素 2 は 3 番ヘッダの未使用要素リストにつながる．　**[例終]**

3.2　未使用要素の比較回数を削減する XCheck

遷移集合が当てはまる位置を探す際，従来手法において節点の候補となる未使用要素リストは 1 本である．それに対して提案手法は，未使用状況に基づいて分類した複数本の未使用要素リストを持ち，それらの先頭を管理するヘッダを持つ．遷移集合が当てはまる未使用状況を計算して，未使用要素リストのヘッダ番号を決定することで，節点候補の要素数を抑制する．

提案手法の変更版 XCheck を図 9 に示す．節点候補を選択するために，未使用要素リストのヘッダ番号 h を図中手順 (XP-1) で後述の関数 TransSetToHead により決定する．節点候補の決定後は従来手法と同様である．

遷移集合が当てはまる未使用状況を求め，対応する未使用要素リストのヘッダ番号を選択する関数 TransSetToHead を図 10 に示す．遷移集合 s に含まれる遷移種のうち，内部表現値が最小の遷移種を s_{min} と定義する．図中手順 (T-2) で s_{min} と s_{min} 以外の遷移種との距離の集合 s' を求める．距離集合 s' は，未使用要素 r に s_{min} を当てはめた際，r から他の未使用要素への相対距離の集合である．すなわち，s' は遷移集合 s が当てはまる未使用状況を表す．最後に (T-3) で，あらかじめ分類された未使用要素リストから s' に対応するヘッダを決定する．これにより，未使用要素を比較する回数の抑制が期待できる．

[例 5] 遷移集合 $s = \{$'a', 'c', 'd'$\}$ から TransSetToHead により未使用要素リストのヘッダを選択する様子を図 11 に示す．s に含まれる遷移種のうち，最小の遷移種は $s_{min} =$ 'a' $= 1$ であるため，(T-2) において他の遷移種 'c' $= 3$, 'd' $= 4$ から $2(= 1+1)$ を引いて $s' = \{1, 2\}$ となる．したがって，(T-3) より選択するヘッダ番号 h は $(011)_2 = (3)_{10}$ となる．　**[例終]**

関数：TransSetToHead
引数：遷移集合 s
戻り値：未使用要素リストのヘッダ番号 h

(T-1) 遷移集合 s に含まれる遷移種のうち，
　　最小のものを s_{min} と定義する
(T-2)s に含まれる s_{min} 以外の遷移種に対して
　　$s_{min}+1$ を引いた遷移種の集合を s' と定義する
(T-3)s' に含まれる遷移種をビット位置と見なし
　　それらのビットを立てた値 h を返す

図 10　関数 TransSetToHead

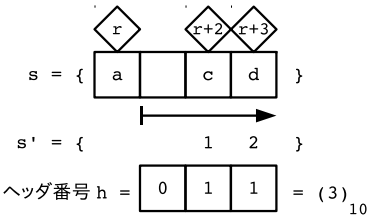

図 11　TransSetToHead の例

提案手法は分類された未使用要素から節点候補を選択するため，従来手法よりも早く遷移集合が当てはまる未使用要素に到達すると考えられ，XCheck が比較する未使用要素個数の減少が期待できる．変更版 XCheck が従来の XCheck に比べて比較回数が減少する例を以下に示す．

[例 6] 遷移が衝突した状況である図 3 のダブル配列に対して提案する未使用要素リストは図 12 となる．ただし，$m=3$ とする．例として，未使用である 2 番要素の未使用状況は $(001)_2=(1)_{10}$ であるため，1 番ヘッダにつながっている．図 12 を対象に，変更版 XCheck で遷移集合 $s=\{$'a', 'b', 'd'$\}$ が当てはまる bse を算出する．はじめに (XP-1) で TransSetToHead により遷移集合から未使用要素リストを選択する．遷移集合 s の中で内部表現値が最小の遷移種 s_{min} は 'a' $=1$ であり，s に含まれる s_{min} 以外の遷移種に対して $s_{min}+1=2$ を引いて，距離集合 $s'=\{0,2\}$ を得る．s' をビット位置とみなしてヘッダ番号 h は $(101)_2=(5)_{10}$ となる．よって，変更版 XCheck による比較の候補は 5 番ヘッダからつながる未使用要素に絞られる．続いて，選択した未使用要素リストから遷移集合 s が当てはまる bse を算出する．(XP-2) で節点候補 $node$ は未使用要素リストの先頭である 9 番要素であり，bse は 8 となる．遷移種 'a' は $node$ に対応し，遷移種 'b' に対応する遷移先として 10 番要素と遷移種 'd' に対応する遷移先として 12 番要素の両方が未使用要素であるため，遷移集合 s が当てはまる $bse=8$ を算出する．以上より，提案手法が bse を算出するまでに比較した未使用要素は 9 のみとなる．ここで，同じ状況で従来手法の XCheck がたどった未使用要素は 3 個であるため，従来手法よりも少ない比較回数で bse を算出する．　**[例終]**

4. 未使用要素管理法の評価

提案手法の XCheck は衝突を解決する際に呼び出されるため，衝突回数を変化させた実験により提案手法を評価する．ここで，字種数が多ければ他の遷移と衝突する候補が増えることから，字種数による衝突回数の変化が予想される．そのため本章では，はじめに字種数の制限による衝突回数の変化を確認する．続いて，字種数を変動させた際の XCheck における未使用要素の比較回数と構築時間を評価する．次に，XCheck の比較回数が提案手法により減少することを確認した後，領域計算量を評価し，最後に，構築以外の操作として探索と削除の性能を確認する．

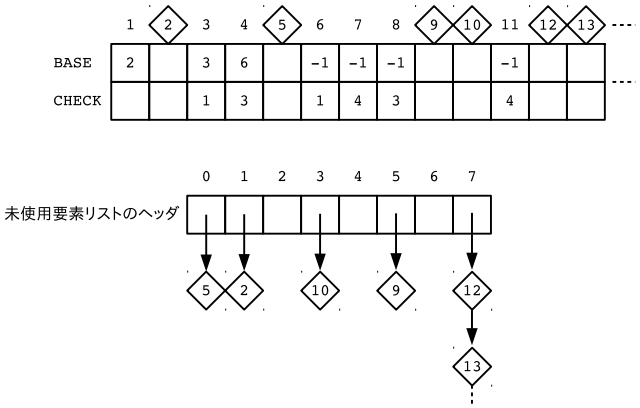

図 12　図 3 のダブル配列に対する提案の未使用要素リスト

表 1　字種の制限

制限	字種数
英小文字	26
英字	52
英数字	62
アスキー文字	95

実験環境として Intel(R) Core(TM) i7 920 @ 2.67GHz，CentOS6.6 32bit，キー集合として英語版 Wikipedia [11] のタイトル集合に対して，表 1 に示す字種数を制限した集合を使用し，ランダムに抽出した 10 万件から 100 万件を昇順に登録した後，ランダムに抽出した 50 万件をランダム順に検索してから削除する．文字コードは UTF-8 を使用する．従来手法として，未使用要素の集合を 1 本の双方向リンクリストで管理する中村らの手法 [10] を使用する．提案手法が未使用状況を見る長さ m を 3 と定義する．

字種数の制限による衝突回数の変化を実証する．従来手法で衝突回数を計測した結果を表 2 に示す．20 万件と 40 万件では字種数別の衝突回数に大きな差は見られなかったが，60 万件から 100 万件では字種数が 26，52，62，95 の順に衝突回数が増加し，字種数 95 の衝突回数は他の字種数に比べて 10 %〜14 %の増加が見られた．すなわち，字種数が多いほど衝突回数が増えている．ここで 100 万件に限定すると，字種数 52 は字種数 26 に比べて 2 %，字種数 62 は字種数 52 に比べて 0.8 %衝突回数が増加した．それに対して字種数 95 は字種数 62 に比べて 10 %も衝突回数が増加しており，字種数 95 と他の字種数との間で衝突回数に大きな差が見られる．そのため，以降の実験では字種数 26，52，62 を区間 A，字種数 95 を区間 B と分けて評価する．

表 2　字種数と衝突回数

件数＼字種数	26	52	62	95
200,000	74,788	73,003	73,079	71,979
400,000	140,036	139,517	140,600	142,775
600,000	200,710	201,964	204,113	214,025
800,000	256,876	261,674	264,145	285,691
1,000,000	309,472	318,430	321,028	355,752

表 3　総比較回数

字種数	26		52	
件数＼手法	提案	従来	提案	従来
200,000	282,563	4,419,915	291,870	4,425,570
400,000	553,944	4,573,909	580,780	6,414,890
600,000	840,563	7,354,258	887,333	10,504,112
800,000	1,166,292	9,678,889	1,253,183	14,633,400
1,000,000	1,590,716	12,168,226	1,805,092	21,154,774
字種数	62		95	
件数＼手法	提案	従来	提案	従来
200,000	295,400	6,419,632	240,292	710,586
400,000	601,832	8,120,901	471,488	1,252,855
600,000	911,668	13,355,710	694,741	1,824,417
800,000	1,266,421	20,233,799	920,887	2,330,676
1,000,000	1,674,356	29,400,126	1,141,516	2,841,403

構築全体を通して XCheck が比較した未使用要素の個数を総比較回数と定義する．提案手法は未使用要素を 2^m 通りに分類するため，従来手法に比べて提案手法の総比較回数は $\frac{1}{2^m}$，つまり減少率は $1-\frac{1}{2^m}$ と予想される．実験では，未使用状況を見る長さ m は 3 であるため，減少率は $1-\frac{1}{2^3}=87.5$ %と予想される．字種数を変動させた際の総比較回数と構築時間を計測した結果をそれぞれ表 3，表 4 に示す．区間 A において，総比較回数は 87〜95 %減少し，構築時間は 18〜31 %の短縮が見られた．区間 B において，総比較回数は 60〜66 %減少し，構築時間は 7〜9 %の短縮が見られた．字種数毎に両手法を比較すると，全ての件数において提案手法が高速となった．ここで，構築時間の短縮率と総比較回数の減少率との相関を考える．相関係数が最小となった字種数 26 における短縮率と減少率を表 5 に示す．このとき相関係数は 0.86 となり，正の相関が見られた．すなわち，総比較回数を抑制することで，構築時間の短縮が期待できる．以上の結果から，未使用状況に基づく未使用要素リストの導入により総比較回数を抑制することで，ダブル配列を高速に構築することを示した．

総比較回数が減少した理由を確認する．構築において XCheck を呼び出した回数を呼び出し回数，XCheck1 回あたりの比較回数を平均比較回数と定義すると，式 (5) が成り立つ．総比較回数，呼び出し回数，平均比較回数について，従来手法に対する提案手法の各減少率を表 6 に示す．ただし，登録件数を 100 万件とする．表 6 より，呼び出し回数の減少率に比べて平均比較回数の減少率が大きく，総比較回数が減少した理由が平均比較回数であると考えられる．

$$\text{総比較回数} \,/\, \text{呼び出し回数} = \text{平均比較回数} \tag{5}$$

表 4　構築時間（秒）

字種数	26		52	
件数＼手法	提案	従来	提案	従来
200,000	0.249912	0.321651	0.245763	0.317102
400,000	0.470479	0.575463	0.470228	0.595360
600,000	0.675497	0.841872	0.679997	0.884016
800,000	0.867668	1.089430	0.878066	1.167520
1,000,000	1.054940	1.328300	1.073440	1.465580
字種数	62		95	
件数＼手法	提案	従来	提案	従来
200,000	0.244463	0.335549	0.266909	0.293056
400,000	0.471728	0.617256	0.533819	0.577212
600,000	0.683346	0.922960	0.800628	0.864219
800,000	0.884416	1.238060	1.063940	1.149420
1,000,000	1.081240	1.571710	1.330950	1.430630

表 5　構築時間の短縮率と総比較回数の減少率

件数	100,000	200,000	300,000	400,000	500,000
構築時間	0.375302	0.223034	0.193772	0.182434	0.192729
総比較回数	0.978872	0.936070	0.907085	0.878890	0.889720
件数	600,000	700,000	800,000	900,000	1,000,000
構築時間	0.197625	0.194454	0.203558	0.200625	0.205797
総比較回数	0.885704	0.874579	0.879501	0.869768	0.869273

表 6　総比較回数，呼び出し回数，平均比較回数の減少率

字種数	総比較回数	呼び出し回数	平均比較回数
26	86.9 %	8.8 %	85.7 %
52	91.5 %	9.3 %	90.6 %
62	94.3 %	9.3 %	93.7 %
95	59.8 %	4.9 %	57.8 %

領域計算量を評価する．評価指標として占有率を式 (6) に定義する．占有率が 1 に近いほど，ダブル配列上で使用要素が凝集しており，コンパクトにトライ木を実現していることを表す．字種数を変動させた際の占有率の計測結果を表 7 に示す．従来手法に比べて，提案手法の占有率は 7〜9 %程度の低下が見られた．提案手法は遷移集合から計算した未使用状況に応じて選択した未使用要素リスト以外に含まれる未使用要素を比較しない．そのため提案手法は，より凝集した配置になる未使用要素を選択できない可能性があり，全ての未使用要素を 1 本のリストで管理する従来手法に比べて占有率が低下したものと考えられる．

$$\text{占有率} = \frac{\text{使用要素の個数}}{\text{最大の使用要素番号}} \tag{6}$$

最後に，構築以外の操作である探索と削除の性能を確認する．探索時間と削除時間の実験結果をそれぞれ表 8，表 9 に示す．探索時間は 0〜10 %の増加が見られた．提案手法と従来手法ではダブル配列の探索アルゴリズムに差はないが，未使用要素の管理方法が異なる．そのため，同じキー集合を登録しても，ダブル配列における要素の配置構造は異なり得る．ここで，ダブル配列は配列内の要素配置によって，探索性能が変化し得る [12]．今回の実験では占有率が低下しており，提案手法により要素の配置構造は変化したため，探索時間に影響したものと

表 7　占　有　率

字種数	26		52	
件数＼手法	提案	従来	提案	従来
200,000	45.01 %	52.49 %	45.75 %	53.97 %
400,000	45.15 %	52.67 %	45.92 %	54.25 %
600,000	45.08 %	52.82 %	45.97 %	54.45 %
800,000	45.01 %	52.98 %	45.92 %	54.55 %
1,000,000	44.88 %	53.07 %	45.90 %	54.67 %
字種数	62		95	
件数＼手法	提案	従来	提案	従来
200,000	48.00 %	56.68 %	85.74 %	94.66 %
400,000	47.87 %	56.68 %	86.48 %	94.74 %
600,000	47.77 %	56.76 %	86.91 %	94.80 %
800,000	47.80 %	56.96 %	87.16 %	94.74 %
1,000,000	47.63 %	56.98 %	87.31 %	94.70 %

表 8　探索時間（秒）

字種数	26		52	
件数＼手法	提案	従来	提案	従来
200,000	0.047643	0.045993	0.047393	0.045943
400,000	0.056592	0.053992	0.057391	0.053492
600,000	0.066940	0.063041	0.067890	0.062441
800,000	0.075789	0.072139	0.075839	0.071439
1,000,000	0.081938	0.079188	0.081838	0.078588
字種数	62		95	
件数＼手法	提案	従来	提案	従来
200,000	0.046943	0.045543	0.069639	0.068939
400,000	0.057691	0.052892	0.089986	0.087637
600,000	0.068639	0.061890	0.109183	0.107283
800,000	0.075289	0.071089	0.123631	0.122531
1,000,000	0.081538	0.078138	0.134030	0.133780

考えられる．

削除時間は 0〜7 %程度の増加が見られた．提案手法と従来手法との間で削除アルゴリズムに差はないが，提案は節点の削除にともない未使用要素を分類するため，削除時間が増加したと考えられる．また，キーを削除する際はダブル配列上の要素を遷移する必要があるため，探索同様に要素の配置構造の変化が削除性能に影響したとも考えられる．

5. おわりに

本稿では，ダブル配列構築の高速化を目的として，未使用要素を右方向に隣接する未使用状況に基づいて管理することで遷移集合が当てはまる基底値の算出を高速化する手法を提案した．実験から，基底値の算出に要する比較回数を削減し，構築時間を高速化した．領域の評価として，ダブル配列全体に対する使用要素の割合である占有率は低下した．構築以外の操作として，探索時間と削除時間は増加が見られた．

今後の課題として，未使用状況を見る長さと性能との関係の分析，未使用状況の包含関係を考慮した分類方法，および探索，削除性能が低下した原因の分析が挙げられる．

表 9　削除時間（秒）

字種数	26		52	
件数＼手法	提案	従来	提案	従来
200,000	0.201519	0.196270	0.193421	0.188371
400,000	0.359495	0.347547	0.348947	0.331700
600,000	0.534369	0.506123	0.513772	0.482627
800,000	0.706193	0.676197	0.675247	0.646052
1,000,000	0.867818	0.836873	0.830274	0.800228
字種数	62		95	
件数＼手法	提案	従来	提案	従来
200,000	0.181672	0.177123	0.089836	0.089436
400,000	0.331350	0.311253	0.132580	0.129580
600,000	0.487176	0.453281	0.175623	0.172474
800,000	0.632204	0.604558	0.215817	0.213118
1,000,000	0.778382	0.750286	0.252662	0.249962

文　献

[1] 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.

[2] 青江順一，森本勝士：ダブル配列法によるトライ検索の実現法，情報処理学会研究報告自然言語処理，NL-085，pp.9-16，1991.

[3] MeCab：http://mecab.googlecode.com/svn/trunk/mecab/doc/index.html，2015/12/27 参照.

[4] ChaSen：http://chasen-legacy.osdn.jp/，2015/12/27 参照.

[5] 蔵満琢麻，望月久稔，ダブル配列を用いた AC マシンにおける遷移の分岐別管理による効率的な辞書構造の実現，情報処理学会：データベース，Vol.2，No.2 (TOD 42)，pp.96-109，2009.

[6] 蔵満琢麻，重越秀美，望月久稔，ダブル配列の遷移拡張による LR 構文解析表の実現と解析速度の高速化，情報処理学会：データベース，Vol.3，No.2 (TOD 46)，pp.80-90，2010.

[7] 森田和宏，泓田正雄，大野将樹，青江順一：ダブル配列における動的更新の効率化アルゴリズム，情報処理学会論文誌，Vol42，No9，pp.2229-2238，2001.

[8] 大野将樹，森田和宏，泓田正雄，青江順一：ダブル配列による自然言語辞書の高速更新法，言語処理学会第 11 回年次大会発表論文集，pp.745-748，2005.

[9] 矢田晋，大野将樹，森田和宏，泓田正雄，吉成友子，青江順一：接頭辞ダブル配列における空間効率を低下させないキー削除法，情報処理学会論文誌，Vol.47，No.6，pp.1894-1902，2006.

[10] 中村康正，望月久稔：圧縮ディジタル探索木における辞書情報更新の高速化手法，情報処理学会:データベース，Vol.47，No.SIG 13 (TOD 31)，pp.16-27，2006.

[11] Wikipedia：https://www.wikipedia.org/,2015/12/18 参照.

[12] 村山智也，望月久稔，遷移先節点に着目したダブル配列構造による探索の高速化の提案，第 13 回情報科学技術フォーラム (FIT2014)，RD-001，pp.1-6，2014.