社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2006－NL－173 (5)
2006／5／19

シングル状態を利用したダブル配列における動的追加の高速化

永井 弘之[†] 藤田 茂[††] 菅原 研次[†††]

千葉工業大学大学院情報科学専攻[†] 千葉工業大学情報工学科[††] 千葉工業大学情報ネットワーク学科[†††]

自然言語処理システムに広く用いられているキー検索法として，トライ法があり，トライを表現可能なデータ構造として，ダブル配列がある．ダブル配列は，検索の高速性と空間利用率の高さを兼ね備えた，優れたデータ構造である．しかし，ダブル配列ではキーの検索時間に比べ，動的追加時間が遅い欠点がある．ダブル配列に対して，キーの動的追加を行うと，衝突が発生し，その回避に多くの計算量を要している問題がある．本論文では，ダブル配列において，遷移可能な次状態が単一であるシングル状態の多数性，およびシングル状態からの遷移先であるシングル要素の機動性を利用し，キーの動的追加時に生じる衝突を，効率的に回避することで，動的追加処理を高速化する手法を提案する．評価実験では，それぞれ10万件のデータを使用し，WordNet英語単語辞書で1.9倍，IPADIC日本語単語辞書で8.7倍，郵便番号で32.5倍，森田らの手法よりも高速に追加できることを確認した．

Efficient Dynamic Key Insertion Algotithms by using Single State for a Double-array structure

HIROYUKI NAGAI[†], SHIGERU FUJITA[††], and KENJI SUGAWARA[†††]

Graduate School of Information and Computer Science[†], Department of Computer Science[††], Department of Information and Network Science[†††], Chiba Institute of Technology

Trie is a well known key retrieve method for natural language processing systems and the Double-array is a fast and compact data structure for a trie. However, dynamic key insertion time is not as fast as key search time, because of resolving collisions take a lot of time. A double-array has many single states and its successor is single elements. Single elements have a property that easy to reallocate. In this paper, we propose a efficient key insertion method by reallocating single elements to resolve collisions. The experimental results for 100 thousand keys, it turned out that the propose method is 1.9 to 32.5 times faster than Morita's method.

1 はじめに

トライ法は，主に自然言語処理システムの辞書構造として広く用いられているキー検索法であり，トライ構造を表現可能なデータ構造として，青江により提案された，ダブル配列 [1] がある．ダブル配列は，2つの配列を利用することで，検索の高速性と，空間利用率の高さを両立させた，優れたデータ構造である．

ダブル配列は，最初にキー集合を構成した後，キーの更新がほとんど行われない準静的検索法として提案されたため，キーの検索時間に比べ，動的追加時間が遅いという欠点がある．

森田らは未使用要素リスト法 [2] を提案し，キーの更新時間がダブル配列の大きさに依存していた問題を解決することで，ダブル配列を動的検索法として確立した．

ダブル配列においては，自然言語からなるキーを格納した場合，遷移可能な次状態が単一である状態 (シングル状態) が75%程度以上を占め，また，シングル状態の遷移先 (シングル要素) は，未使用要素に容易に移動できる特徴を持っている．大野ら [3] は，シングル状態の特徴を利用することで，高い空間利用効率を保つことができる，高速な削除法を提案している．

本論文では，動的追加を行う際に，約85%以上の高頻度で衝突が発生していること，また，衝突回避処理が動的追加時間の大部分を占めていることに着目し，シングル要素の特徴を利用することで，衝突回避処理を効率的に行い，動的追加時間を高速化する手法を提案する．

提案手法の有効性を確認するために，評価実験を行った．実験では，WordNet英語単語辞書 [4]，IPADIC日本語単語辞書 [5]，7桁郵便番号より各10万件のデータを使用し，提案手法が森田らの手法に比べ，1.9～32.5倍高速に追加できることを確認した．

2 ダブル配列

2.1 トライとダブル配列

キーを構成する各記号を遷移ラベルとし，各キーの共通接頭辞部分を併合して構成される木構造であるトライは，共通接頭辞や検索失敗位置の検出が容易にできる．また，各状態の遷移が一定時間でできるならば，キーの探索時間計算量は，トライに格納されているキー総数に依存せず，そのキーの長さにのみ比例するため，高速な検索が可能である．これ

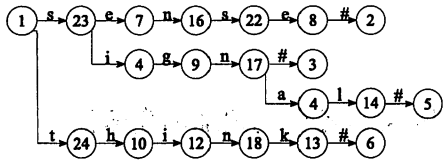

図 1: キー集合 K を保持するトライ

らの特徴を持つことから，トライ法は，主に自然言語処理システムの辞書構造に広く用いられている [6].

トライの初期状態を 1 とし，トライの葉とキーを 1 対 1 に対応させるために，キーを構成する文字集合には含まれない記号 '#' を，それぞれのキーの終端に付加することとする.

キー集合 K = {sense#, sign#, signal#, think#} を格納したトライの例を，図 1 に示す.

トライを表現できるデータ構造に，ダブル配列がある．ダブル配列では，BASE と CHECK と名付けられた 2 つの配列を使用する．トライにおいて，ある状態 s から，次状態ノード t にラベル l で遷移可能であるとき，ダブル配列では次の式 1, 2 を満たす．ここで，N(l) は，ラベル l を内部コード値に変換する関数である.

$$t = BASE[s] + N(l) \quad (1)$$
$$CHECK[t] = s \quad (2)$$

キー集合 K に対し，森田らの手法 [2] に基づいてダブル配列を構築した例を，図 2 に示す．ただし，未使用要素リストは正の値を用いており，未使用要素リストの先頭インデックス番号は，$eHead$ により管理されている.

このとき，各記号の内部コード値は，終端記号 '#' = 1，文字 'a'〜'z' を 2〜27 に対応させている．また，トライの状態番号と，ダブル配列のインデックス番号は 1 対 1 に対応させることができるため，以下同等として扱う．また，トライの葉に対応する，ダブル配列のインデックスの BASE 値は負であり，未使用インデックスの BASE 値は 0 とする.

トライとダブル配列の対応例を示す．トライの状態番号 23 から，'e' により定義されている遷移について，まず，式 1 により遷移先 t = BASE[23] + N('e') = 1 + 6 = 7. 式 2 について，CHECK[7] = 23 であるので，線が定義されていることがわかる．また，状態番号 23 から，遷移が定義されていない 'f' についてみると，遷移先 t = BASE[23] + N('f') = 1 + 7 = 8 であるが，CHECK[8] = 22 $\neq$ 23 であるため，遷移が定義されていないことがわかる.

2.2 検索アルゴリズム

ダブル配列の最大インデックス番号を MAX とするとき，キー $X = k_1k_2 \ldots k_nk_{n+1}, k_{n+1}$ ='#' に対する検索が成功すれば TRUE を，失敗すれば FALSE を返す関数 DA_SEARCH(X) のアルゴリズムを，以下に示す.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BASE	3	-1	-1	1	-1	-1	1	1	2	2	1	3	5	4	0	2	2	1	0	0	0	2	1	1
CHECK	25	8	17	17	14	13	23	22	11	24	23	10	18	4	19	7	9	12	20	21	25	16	1	1

eHead=15

図 2: キー集合 K を格納したダブル配列

[関数 DA_SEARCH(X)]

手順 (S-1): 変数の初期化

ダブル配列の現在のインデックス番号 s，およびキーの文字位置 pos を 1 に初期化する.

手順 (S-2): 状態遷移の確認

t =BASE[s]+N(l) により，次インデックス番号 t を求め，0 < t <MAX+1，CHECK[t]= s が成り立つか確認する．成り立たない場合，検索は失敗したので，FALSE を返して終了する．成り立つ場合は，インデックス番号 s から t への遷移が定義されているので，t を s に代入し，pos に 1 を加える.

手順 (S-3): 終了判定

$BASE[s] < 0$ であれば，インデックス s はトライの葉ノードに対応するので，TRUE を返して終了する．そうでなければ，手順 (S-2) に戻る.

2.3 シングル状態とマルチ状態

シングル状態とは，トライのある状態が兄弟を持たない場合，すなわち，遷移可能な次状態が単一であるか，存在しない場合の状態をいう．マルチ状態は，シングル状態でない状態のことをいう.

図 1 の例で示すと，状態番号 2，9 はシングル状態であり，状態番号 17，23 はマルチ状態である.

シングル状態 s から遷移できる状態をシングル要素という．シングル状態の定義より，このとき s から遷移可能な次状態は，ただ 1 つである．ラベル l により遷移が定義されるシングル要素は，ダブル配列において，N(l) 以上の未使用インデックスに必ず移動することができるという特徴を持つ．また，マルチ状態から遷移可能な次状態をマルチ要素という.

大野ら [3] は，ダブル配列のあるノード s について，CHECK[|CHECK[s]|] > 0 *ならば s はシングル要素であり，CHECK[|CHECK[s]|] < 0 ならば s はマルチ要素であると定義することで，シングル要素，マルチ要素の判別を容易とする手法を提案している．大野らの手法により，図 2 の CHECK 値を再定義したものを，図 3 に示す.

図 3 のダブル配列で例を挙げる．ノード 9 について，CHECK[|CHECK[9]|] = CHECK[11] = 23 > 0 であるから，シングル要素であることがわか

*|s| は s の絶対値を示す

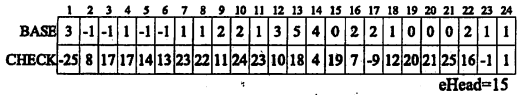

図 3: 大野らの手法を適用したダブル配列

る．ノード 4 について，CHECK[|CHECK[4]|] = CHECK[17] = −9 < 0 であるから，マルチ要素であることがわかる．

3 動的追加アルゴリズムと問題点

3.1 動的追加アルゴリズム

ダブル配列に動的にキーを追加するアルゴリズムとして，森田らが提案した未使用要素リスト法がある．未使用要素リスト法では，新たな遷移を定義する要素を，未使用要素のみ確認可能としたことで，ダブル配列のサイズに依存しない動的追加を可能にした．

未使用要素リスト法では，ダブル配列の未使用インデックス番号が，昇順に $e_1, e_2, \ldots, e_n$ であるとき，

$$\begin{aligned} CHECK[e_i] &= e_{i+1} \\ CHECK[e_n] &= MAX + 1 \end{aligned}$$

とし，未使用要素リストの先頭インデックス番号 e_1 は，変数 $eHead$ で表すことで，未使用要素リストを構成する．未使用要素リストにより，新たな遷移の定義を行う際に，未使用要素のみ確認していくことが可能となる．

動的追加アルゴリズムで使用する関数を，以下に説明する．

関数 GET_LABELS(s)
ある状態 s から次状態に遷移可能であるラベルの集合を求める．

関数 W_BASE(idx, val)
BASE[idx] に val を書き込む．必要に応じて，ダブル配列を拡張する．

関数 W_CHECK(idx, val)
CHECK[idx] に val を書き込む．idx が未使用であったならば，未使用要素リストから idx を除去する．val が -1 のときは，ダブル配列のノード idx を未使用とし，idx を未使用要素リストに追加する．必要に応じて，ダブル配列を拡張する．

関数 X_CHECK(R)
遷移ラベルの集合 R について，それらの遷移先インデックス t が全て未使用となる，最小の BASE 値 q ($q \geq 1$) を，未使用要素リストをたどることで求める．

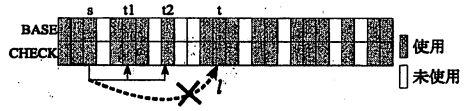

図 4: 衝突の発生

ダブル配列に，キー X を動的に追加する場合，まず，関数 DA_SEARCH(X) により検索を行う．そして，手順 (S-2) における検索失敗インデックス番号 s，キー位置 pos より，関数 DA_INSERT(s, pos) を呼ぶことで，追加が行われる．関数 DA_INSERT(s, pos) のアルゴリズムを，以下に示す．

[関数 DA_INSERT(s, pos)]
手順 (I-1): 衝突の判定と回避
CHECK[BASE[s]+N(k_{pos})] > 0 であれば，衝突が発生しているので，関数 RESOLVE(s, k_{pos}) により，衝突の回避を行う．
手順 (I-2): 遷移先の定義
遷移先 t = BASE[s]+N(k_{pos}) を求め，W_CHECK(t, s) により，遷移を定義する．ここで，ノード s に分岐が生じたことになるから，s をマルチ状態とする．残りの文字列の遷移を定義するために，t を s に代入し，pos に 1 を加える．
手順 (I-3): 残りの文字列の遷移を定義
W_BASE(s, X_CHECK(k_{pos})) により BASE 値をセットし，遷移先 t = BASE[s]+N(k_{pos}) を求め，W_CHECK(t, s) により遷移を定義する．次の遷移を定義するために，t を s に代入し，pos に 1 を加える．
手順 (I-4): 終了判定と終端定義
$pos > n + 1$ であれば，全ての文字について定義が終了したので，W_BASE(t, -1) によりトライの葉と対応させ，終了する．そうでなければ，残りの文字列を追加するために，手順 (I-3) に戻る．

3.2 衝突回避

ダブル配列において，動的にキーを追加するとき，ある状態に新たな子を追加する必要が生じる．このとき，新たな子の定義先インデックスが，既に他の遷移の定義により，使用されており，遷移が定義できない場合が起こりうる．これを衝突という．

衝突が生じたときの例を，図 4 に示す．図 4 では，インデックス番号 s から，既に $t1$, $t2$ への遷移が定義されている．このとき，新たにラベル l によって遷移が，s から t に定義されようとしたとき，t が既に他の遷移定義に使用されているため，衝突が発生する．

森田らの手法では，衝突が発生した場合，親から遷移可能な全てのラベルに，新たに遷移を定義するラベルを加えたものについて，それら全ての遷移先

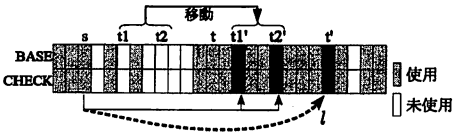

図 5: 森田らによる衝突回避

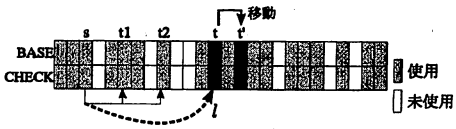

図 6: シングル状態を利用した衝突回避

が未使用となる BASE 値を求め，それらの遷移を移動することにより，衝突の回避を行う．その様子を図 5 に示す．

森田らの手法に基づいて，インデックス番号 s から，ラベル l による遷移の定義を試みたとき，発生する衝突を回避する関数 RESOLVE(s, l) のアルゴリズムを，以下に示す．また，衝突回避のために行われるノードの移動と，それにともなって必要となる CHECK 値の再定義を行う関数 MODIFY のアルゴリズムを以下に示す．

[関数 RESOLVE(s, l)]

手順 (R-1): 遷移ラベルの収集

GET_LABELS(s) により，s から遷移可能なラベルの集合 R を得る．

手順 (R-2): BASE 値の探索

X_CHECK($R \cup \{l\}$) により，R に l を加えた場合でも，遷移先が未使用となる s の BASE 値 *new_base* を求める．

手順 (R-3): 状態の移動と再定義

手順 (R-1) で得られた s から遷移可能なラベルの集合 R，手順 (R-2) で得られた新たな BASE[s] = *new_base* に基づいて，既存の遷移先の移動，および遷移先を移動したことにより必要となる，遷移先の遷移先の遷移元を示す CHECK 値の再定義を，MODIFY(s, *new_base*, R) により行う．

[関数 MODIFY(s, *new_base*, R)]

手順 (M-1): 新しい BASE 値の定義

BASE[s] の値を *old_base* に保存し，W_BASE(s, *new_base*) により，衝突を回避できる新しい BASE 値を定義する．

手順 (M-2): 遷移の移動

新しい BASE 値に基づいて，ノードの移動を行う．R に含まれるそれぞれのラベル l について，以下の処理を行う．新しい遷移先インデックス番号 t = BASE[s] + N(l) を求め，W_CHECK(t, s) により遷移を定義する．*old_t* = *old_base* + N(l) に対して，W_BASE(t, BASE[*old_t*]) により，元の遷移先にある BASE 値を移動先にコピーする．このとき，BASE[*old_t*] > 0 であれば，*old_t* からの遷移先の CHECK 値を再定義する必要があるので，手順 (M-3) を行う．最後に，インデックス番号 *old_t* を未使用にして，終了する．

手順 (M-3): 遷移先の遷移先の再定義

CHECK[q] = *old_t* となる全ての q に対して，W_CHECK(q, t) により，遷移を再定義する．

3.3 森田らの手法の問題点

森田らの手法では，衝突が発生した場合，新たに子を追加しようとしている親の BASE 値を，新たな子を加えた全ての子の遷移先インデックスが未使用となるよう変化させることで，衝突を回避している．

しかしこの手法では，既に定義されている遷移ラベルに，これから定義しようとしている遷移ラベルを加えたものについて，衝突が回避できる BASE 値を求め，それらを移動させる必要が生じる．また，それぞれについて遷移先の再定義が必要になるため，衝突を回避するために必要な計算量が大きくなり，その結果，動的追加時間を増大させてしまっている問題がある．

森田らの動的追加アルゴリズムに対する改良手法として，中村ら [7] により，未使用要素リストを双方向にし，未使用要素リストの探索，および更新の高速化手法が提案されているが，衝突回避アルゴリズムについての検討はなされていない．

4 提案手法

4.1 提案手法の概要

森田らの手法では，新たに遷移を定義しようとしている親の BASE 値に注目し，遷移が定義されているラベル全て，および新たに遷移を定義するラベルについて，それらの遷移先が未使用となる BASE 値を求め，それに基づいて，定義されている子ノード全てを移動させることで，衝突を回避する．

しかし，衝突の原因は，インデックス t を使用している状態とも考えられる．つまり，t に既に定義されている遷移を移動することができれば，衝突を回避できたことになる．このとき，t に定義されている要素がシングル要素であれば，その遷移は未使用要素 t' へ，容易に移動することができる．その様子を図 6 に示す．

提案手法では衝突が発生した場合，新たに遷移を定義しようとしたインデックスにシングル要素が格納されていれば，そのシングル要素を移動させることで，衝突の回避を行う．また，既に定義されている状態がマルチ状態であれば，森田らの手法を用い

た衝突解消を行う．シングル要素の移動と再定義は，常に一定時間で行うことができるため，衝突回避に要する時間が短縮でき，動的追加時間を高速化することができる．

4.2 シングル状態を利用した衝突回避

提案手法では，t の親 s' がシングル状態であるとき，つまり，CHECK[s'] = CHECK[|CHECK[t]|] = CHECK[|CHECK[BASE[s]+ N(l)]|] > 0 であるとき，t に定義されているシングル要素を，t' へ移動することで，t を未使用要素とし，遷移定義を可能とすることで，衝突の回避を行う．なお，t がマルチ要素であるとき，または，$s = t$ となってしまうときは，手順 (R-1) から (R-3) に示した，既存手法による衝突解消処理を行う．

インデックス番号 s から，ラベル l で遷移を定義しようとしたとき，その遷移先 t において衝突が発生し，t を t' に移動することで，衝突を回避する関数 RESOLVE_SINGLE(s, l) のアルゴリズムを，以下に示す．

[関数 RESOLVE_SINGLE(s, l)]

手順 (R'-1): 遷移ラベルの収集

衝突発生インデックス t を式 1 に基づいて，t = BASE[s] + N(l) として求める．t はシングル要素であるから，その遷移元 s' = |CHECK[t]| から遷移可能なラベル l' は単一であり，N(l') = t - BASE[s'] により求められる．

手順 (R'-2): BASE 値の探索

t を移動するために，t の遷移先 t' が未使用要素となる s' の BASE 値 *new_base'* を X_CHECK(l') により探索する．

手順 (R'-3): シングル要素の移動

手順 (R'-1) により求めた l'，手順 (R'-2) により求めた，*new_base'* に基づいて，状態 t に定義されている遷移の移動を，関数 MODIFY(s', *new_base'*, l') により行う．

また，衝突発生インデックスがシングル要素である場合，関数 RESOLVE_SINGLE を，そうでない場合は関数 RESOLVE を呼ぶように，関数 DA_INSERT の手順 (I-1) を次のように変更する．

手順 (I-1): 衝突の判定と回避

t = CHECK[BASE[s]+N(k_{pos})]) > 0 であれば，衝突が発生している．このとき，CHECK[|CHECK[t]|] > 0 であれば，衝突の原因となった既存の遷移は，シングル要素であるので，関数 RESOLVE_SINGLE(s, k_{pos}) により，衝突の回避を行う．そうでない場合，関数 RESOLVE(s, k_{pos}) により，衝突の回避を行う．

5 評価

5.1 理論的評価

本論文では，検索アルゴリズムに対する変更を行っていないので，ここで評価は行わない．

以下，遷移ラベルの総数を e，未使用ノード数を m，使用ノード数を n とする．このとき，森田らの手法により，衝突を回避するために必要な最悪計算量を考える．手順 (R-1) では，関数 GET_LABELS により，遷移可能なラベルを収集するため，$O(e)$ である．手順 (R-2) では，関数 X_CHECK を使用するため，$O(m \cdot e)$ である．手順 (R-3) では，関数 MODIFY を使用するため，$O(m \cdot e + m \cdot e^2)$ となる．

次に，提案手法により，衝突を回避するために必要な最悪計算量を考える．手順 (R'-1) では，シングル状態からの遷移は 1 つであり，遷移元の BASE 値と，遷移先インデックス番号より，ラベルの内部コード値を求められるため，$O(1)$ である．手順 (R'-2) では，関数 X_CHECK を用いるが，このとき渡すラベルは必ず 1 つであるため，$O(m)$ である．最後に，手順 (R'-3) では，関数 MOFIFY を使用するが，ここでも移動し再定義しなければならないラベルは 1 つであるため，$O(m + m \cdot e)$ となる．

5.2 実験的評価

森田らの手法に対する提案手法の有効性を確認するために，評価実験を行った．

提案手法は，約 500 行の C++言語にて実装されており，OpenSuSE Linux 10.0 が動作している，Intel PentiumIII 1133MHz の計算機を用いて実験を行った．実験では，日本語単語辞書として IPADIC [5] より名詞見出し語，英語単語辞書として WordNet [4] より名詞見出し語，それぞれランダムに 10 万語を，キー集合として用いた．また，シングル要素が占める割合が，約 45%と低く，提案手法にとって条件が悪いものとして，郵便番号 10 万件についても実験を行った．遷移ラベルの総数は，1 バイトで表現可能なラベル数 256 に，終端記号を加えた 257 である．

各キー集合を 10 万件追加したときの空間利用状況を，表 1 に示す．また，追加時間に関する実験結果を，表 2 に示す．表 2 における，R_SINGLE は，提案手法により行われた回数である．

表 1: 空間利用状況

	WordNet	IPADIC	郵便番号
キー総数	10,000	10,000	10,000
使用要素数 n	671,326	394,583	222,145
シングル要素数	520,627	266,455	101,401
未使用要素数 m			
既存手法	76	11,749	9,291
提案手法	105	259	47

表 2: 追加時間に関する実験結果

キー数	10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000	90,000	100,000
WordNet										
衝突回数										
既存手法	9,986	19,981	29,981	39,980	49,978	59,977	69,976	79,974	89,973	99,971
提案手法	9,994	19,994	29,993	39,992	49,991	59,991	69,991	79,990	89,990	99,990
R_SINGLE	8,441	16,678	24,844	32,920	41,020	48,911	56,772	64,680	72,525	80,417
追加時間 [ms]										
既存手法	260	506	752	998	1,242	1,489	1,734	1,979	2,223	2,469
提案手法	138	264	395	525	647	774	911	1,031	1,155	1,290
IPADIC										
衝突回数										
既存手法	8,633	17,688	26,644	35,639	44,621	53,512	62,366	71,110	79,737	88,268
提案手法	9,912	19,781	29,546	39,241	48,845	58,358	67,789	77,129	86,344	95,521
R_SINGLE	7,978	15,449	22,729	29,891	36,864	43,693	50,422	56,993	63,436	69,921
追加時間 [ms]										
既存手法	835	2,048	3,644	5,449	7,629	10,089	12,780	15,735	18,957	22,372
提案手法	273	533	790	1,040	1,300	1,560	1,807	2,060	2,322	2,565
郵便番号										
衝突回数										
既存手法	9,898	19,622	29,200	38,637	47,906	56,841	65,323	73,244	80,508	87,001
提案手法	9,913	19,727	29,484	39,224	48,936	58,648	68,292	77,919	87,545	97,149
R_SINGLE	6,831	13,171	19,193	25,128	31,052	37,123	43,601	50,551	57,968	65,886
追加時間 [ms]										
既存手法	188	439	858	1,576	2,791	4,638	7,691	12,207	19,432	31,537
提案手法	103	200	293	395	492	597	708	804	885	971

表 2 より，10 万件のデータについて提案手法は既存手法に対し，WordNet 英語単語辞書で 1.9 倍，IPADIC 日本語単語辞書で 8.7 倍，郵便番号で 32.5 倍の高速化を得られていることがわかる．また，衝突発生回数についてみると，既存手法よりも提案手法が，衝突発生回数が多くなっているが，追加時間は提案手法の方が高速であり，提案した衝突回避アルゴリズムは効率的であると言える．表 1 より，未使用要素数 m は，既存手法と同等か，それ以下の値に抑えられている．これは，衝突回避の際にシングル要素を移動することで，より効率的に未使用要素が使用されているためであると考えられる．

特に，郵便番号について，提案手法により最も高速化の効果がみられた．郵便番号のように機械的に生成されたキー集合は，マルチ状態に対応するマルチ要素が多いため，既存手法における衝突回避方法では，移動および遷移の再定義をしなければならないノード数が多く，衝突回避に要する計算量が増大する．しかし，提案手法では，表 1, 2 に示した結果より，シングル要素が 45%を下回った場合でも，衝突の約 65%が，シングル要素の移動によって回避できていることがわかり，提案手法の有効性を確認できる．

6 おわりに

本論文では，シングル要素の移動が容易にできる特徴に着目し，衝突の回避を効率的に行うことにより，動的追加時間の高速化を可能にした．

また，日本語単語，英語単語，郵便番号，各 10 万件をキー集合として評価実験を行い，提案手法の有効性を確認した．

今後の課題として，動的更新が頻繁に行われる実システム上に，提案手法に基づくダブル配列を実装し，提案手法の有効性を確認することが挙げられる．

参考文献

[1] 青江順一. ダブル配列による高速ディジタル検索アルゴリズム. 電子情報通信学会論文誌 D, Vol. J71-D, No. 9, pp. 1592–1600, 1988.

[2] 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, Vol. 42, No. 9, pp. 2229–2238, 2001.

[3] 大野将樹, 森田和宏, 泓田正雄, 青江順一. ダブル配列におけるキー削除の効率化手法. 情報処理学会論文誌, Vol. 44, No. 5, pp. 1311–1320, 2003.

[4] George A. Miller. Wordnet: a lexical database for the english language. version 2.1. http://wordnet.princeton.edu/.

[5] 奈良先端科学技術大学院大学 情報科学研究科 自然言語処理学講座. Ipadic version 2.7.0. http://chasen.aist-nara.ac.jp/stable/ipadic/.

[6] 青江順一. キー検索技法—トライ法とその応用. 情報処理学会誌, Vol. 34, No. 2, p. 244, 251 1993.

[7] 中村康正, 望月久稔. ダブル配列における動的更新アルゴリズムの高速化. FIT2005 予稿集 D-046, pp. 109–110, 2005.