自然言語処理 85－2
（1991. 9. 19）

ダブル配列法によるトライ検索の実現法

青江　順一　森本　勝士
徳島大学工学部

自然言語処理の辞書検索手法として有効利用されているトライ法を効率的にインプリメントするためのデータ構造として，二つの1次元配列を使用するダブル配列法を提案する．また，自然言語処理の辞書は，予め基本的キー集合を構築しておき，後に使用者がキーを適宜追加して拡充させる場合が一般的であるので，本手法はこの条件を満足する場合（準静的検索法）を満足するものである．本手法では，トライにおいて分岐なしに終端のノードまで連続する遷移列をダブル配列とは別のデータ構造TAILにストリングとして格納し，このダブル配列とTAILを使ったトライ法の検索，追加，削除アルゴリズムを提案する．

キーの長さをk；ダブル配列上で冗長に定義される状態番号数をm；入力記号の種類をeとするとき，検索，削除，追加の最悪の計算時間がそれぞれO(k)，O(m)，O(m・e+e²)となることを示す．また，千個以上のキー集合に対する種々の実験結果から，このmはダブル配列に定義されるべき全状態数に対して，1パーセント以下の小さな値となることが分かった．

IMPLEMENTATION OF TRIE SEARCH STRATEGIES BY DOUBLE-ARRAY STRUCTURES

Jun-ichi Aoe, Katushi Morimoto
The University of Tokushima
Minami-Josanjima-Cho, Tokushima-Shi, 770 Japan

An efficient trie search algorithm is presented in this paper by introducing a new internal array structure, called a double-array, that combines the fast access of a matrix form with the compactness of a list form. Each arc of the trie can be computed from the double-array in O(1) time; that is to say, the worst-case time complexity for retrieving a key becomes $O(k)$ for the length k of that key. The double-array is modified to make the size compact while maintaining fast access and algorithms for retrieval, insertion and deletion are presented. Suppose that the size of the double-array is $n + m$, n being the number of nodes of the trie, e being the number of input symbols, and m being the number of redundant nodes on the double-array. Then it is theoretically proved that the worst-case times of deletion and insertion are proportional to $O(m)$ and $O(me+e^2)$ respectively, independent of n. From the experimental results of building the double-array incrementally for various sets of keys, it is shown that m has an extremely small value, ranging from 0.17 to 1.13.

1．まえがき

自然言語処理システムは，辞書情報を利用した解析が必要不可欠であり，辞書検索効率がシステム全体の能率に大きな影響を与える[9,20,27]．特に，分かち書きされていないストリング処理（かな漢字変換，日本語形態素解析）では，見出し（キー）の切り出しに有効なキー検索手法を利用する必要がある．

キー検索法[6,21,26]は適用分野により，キーの更新が頻繁に行なわれる動的キー集合を対象とする場合（動的検索法と呼ぶ）とこの更新が全く行なわれない静的キー集合を対象とする場合（静的検索法と呼ぶ）とに分類できる．動的検索法ではキーの更新を考慮するために，空間の使用効率あるいは検索効率が低下する場合があるが，静的検索法ではこれらの効率を極めて高い水準におくことが可能となる．動的検索法には，分散記憶法であるハッシュ法；探索木法である2進木法，B⁺木法；ディジタル検索法(digital search，広義には探索木法と考えられる）であるトライ(trie)法[2,6,9,11,13,16,17,19,21-26]，トライハッシュ法，拡張ハッシュ法(15,19)等がある．また，静的検索方法は，完全ハッシュ法[12,14,]が代表的であり，その外トライに対する静的データ構造[22]を提案したものがある．しかし，この分類は極端であって，現実の応用分野では，自然言語処理における辞書の様にキー集合の大部分が予め分かっていて，後で適宜キーを追加して拡充する場合も非常に多い．この様なキー集合に対する検索法を準静的検索法[7]と呼ぶ．

従って，本報告では，ダブル配列法[7-10]を使ったトライ法を準静的検索法として確立するために，ダブル配列の拡張法を提案し，拡張されたダブル配列に対するキーの検索，追加，削除アルゴリズムを提案する．そして，これらアルゴリズムに対する最悪の計算時間を決定し，種々のキー集合に対する実験結果から本手法の有効性を実証する．

2．　トライ検索マシンとダブル配列

2．1　トライ検索マシン

キー集合Kに対するトライ検索マシンMをストリングパターンマッチングマシン[1,3,4,5]の記法で定義する．

$$M = (S, I, g, s_1, U)$$

で表す．ここで，S，Iはそれぞれ状態，入力記号の有限集合；s_1は初期状態（番号1で表す）；gは状態遷移関数でgoto関数と呼ばれる．$U(\subseteq S)$は出力状態の有限集合を表す．以後，状態は整数値の番号を対応させる．関数gは$S\times I$から$S\cup\{fail\}$への写像を定義し，状態sからtへの遷移が記号aに対して定義されていれば，g(s,a)=tと書き，そうでなければg(s,a)=failと書く．以後，状態sに入る遷移の数をindegree(s)，そこから出る遷移の数をoutdegree(s)で表す．そして，outdegree(s)=0なる状態sを最終状態と呼び，その集合をFで表す．また，goto関数gは$S\times I^*$から$S\cup\{fail\}$への写象に拡張して記述される．ここで，I^*はI上のすべてのストリングの集合を表す．以後，a，b，c，$d\in I$；x，y，$z\in I^*$を使用し，空記号列(empty string)をεで表す．以後，キーの最後尾にのみ付加される特別な記号#を定義し，最終状態と出力状態を一致させ，次の定義を行う．

［定義１］ マシンMに対して，次の状態を定義する．

(1) g(s,a)=t，outdegree(s)≧2なる状態sから到達可能な最終状態までの遷移列上にoutdegree(r)≧2なる状態rが存在しないならば，状態sをセパレート状態と呼び，その集合をS_Pで表す．

(2) 初期状態からセパレート状態までの遷移列上に存在する状態（初期状態は含み，セパレート状態は含まない）をマルチ状態と呼び，その集合をS_Mで表す．

(3) セパレート状態から最終状態までの遷移列上に存在する状態（セパレート状態は含み，最終状態は含まない）をシングル状態と呼ぶ．

［定義２］ g(s,y)=t，$s\in S_P$，$t\in F$なるストリングyをセパレート状態sに対するシングルストリングと呼び，STR[s]で表す．

［例１］ キー集合K'={bac#，bc#，ba#，bab#}に対するgoto関数を図１に示す．マルチ状態は1,3,7；シングル状態は4,5,6,9であって，セパレート状態と対応するシングルストリングは次のようになる．

STR[4]=#，STR[5]=#，STR[6]=ε，STR[7]=#．

2．2　ダブル配列

ダブル配列[6-10]はBASEとCHECKの二つの一次元配列を使用し，g(s,a)=tなる遷移を次の様に定義する．

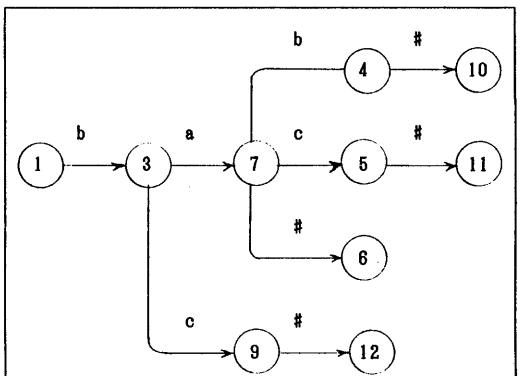

図１　K'に対するトライ

t = BASE[s] + a; CHECK[t] = s

ここで，記号aは1以上の内部表現値(numerical value)として取り扱われている．ダブル配列法は状態番号sと入力記号aの内部表現値との関係を利用して，遷移を確認するべきCHECKのインデックス(しかもこれが次に進むべき状態番号にも一致する)を巧妙に決定する（図2参照）．

本手法では，$S_M \times (S_M \cup S_P)$上の遷移をダブル配列に格納し，セパレート状態以後の遷移は配列TAILにシングルストリングとして格納する．この拡張方法は，次の効果がある．

1) ダブル配列に格納する遷移数が減少するので，配列要素のバイト数を少なくできる．また，ダブル配列では一つの遷移に二つの要素を必要とするが，TAIL上では1記号のバイト数だけに記憶量を軽減できる．

2) シングル状態からの遷移がストリングのマッチングだけで確認できるので，検索を高速化できる．

しかし，この拡張に対しては処理中の状態がセパレート状態であるか否かを判定し，しかもTAIL中のシングルストリングの位置が決定できる効率的手法が必要不可欠となる．そこで，本報告では，次の条件を満足するダブル配列を定義する．

［定義３］　次が成立するならば，ダブル配列は条件Aを満足すると言う．

(A-1)　$g(s,a)=t$; $s \in S_M$; $t \in S_M \cup S_P$ならばそのときに限って次が成立する．

BASE[s] + a = t, CHECK[t] = s.

(A-2) $s \in S_P$なる状態番号sに限って次が成立する．

BASE[s]<0.

(A-3) $STR[s]=b_1 b_2 \ldots b_m (1 \leqq m)$; $s \in S_P$なるセパレート状態s_rに対して，次が成立する．

p=-BASE[s];

$TAIL[p]=b_1$, $TAIL[p+1]=b_2$, ... , $TAIL[p+m-1]=b_m$.

条件(A-2)と(A-3)はBASE[s]が負ならばsはセパレート状態であって，しかも対応するシングルストリングがTAILの位置-BASE[s]からアクセスできることを意味する．また，ダブル配列のインデックスの最大値を越える場合の判定を行なうために，次を定義する．ただし，ダブル配列の未使用要素の値0とする．

［定義４］　次が成立するならばダブル配列は条件Bを満足すると言う．

初期状態番号1に対するCHECK[1]の値は，ダブル配列のCHECK[q]≠0なる最大のインデックスqと一致する．

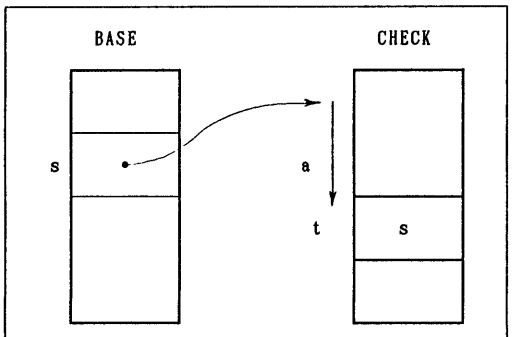

図２　ダブル配列の説明図

３．　ダブル配列によるキー検索

拡張されたダブル配列を使用した検索アルゴリズム１を図３に示す．但し，次の変数と関数を使用する．

S_TEMP: TAILから取り出されたストリングを保持する．

FETCH_STR(p): TAIL[p]からTAIL[p+k]=#,$0 \leqq k$なる記号(#は含む)までのストリングを返す．

STR_CMP(y,z): ストリングyとzが一致すれば-1を，一致しなければ部分マッチした接頭辞の長さを返す．

アルゴリズム１の行(1-1)は，tがCHECKの最大のインデックスを越えるかあるいは遷移が未定義の場合，FALSEを返す．また，repeat文はBASE[s]が負になれば終了し，行(1-2)はシングルストリングSTR[s]をS_TEMPに代入する(ただし，h=n+1ならばマッチングが成功し，TRUEを返す)．そして，行(1-3)でS_TEMPと残りの入力ストリングとの比較によりxがKに含まれるか否かを決定する．

［例２］　キー集合K'に対するダブル配列(次章で構成される)を図４に示す．ただし，記号a,b,c,#の内部コードはそれぞれ1,2,3,4とし，TAILの記号'?'は次章で説明される未使用記号を表す．キーbc#の検索例を示す．

t=BASE[1]+b=3

```
 Algorithm 1.  Retrieval Algorithm.
Input: A string x#=a1a2 ... an an+1, an+1= #;
     and the machine M with the double-array
     and TAIL for K.
Output: If x∈K, then the output is TRUE,
        otherwise FALSE.
Method:
   begin
      s:=1;  h:=0;
      repeat
         h:=h+1; t:=BASE[s]+ah;
(1-1)    if(t>CHECK[1]) or (CHECK[t]≠s)
         then return(FALSE) else s:=t
      until BASE[s]<0;
(1-2) if ah=# then return(TRUE)
      else S_TEMP:=FETCH_STR(-BASE[s]);
(1-3) if STR_CMP(ah+1... anan+1,S_TEMP)=-1
      then return(TRUE) else return(FALSE)
   end.
```

図３　ダブル配列による検索アルゴリズム

	1	2	3	4	5	6	7	8	9
BASE	1	0	6	-8	-1	-7	2	0	-5
CHECK	9	0	1	7	7	7	3	0	3

	1	2	3	4	5	6	7	8	9
TAIL	#	$	?	?	#	$	$	#	$

図４　K'に対するダブル配列

(1-1): t=3≦CHECK[1]=9 且つ CHECK[3]=1
BASE[s]=BASE[3]=6>0, t=BASE[3]+c=9

(1-1): t=9≦CHECK[1]=9 且つ CHECK[9]=3
BASE[s]=BASE[9]=-5<0

(1-2): h=2≠n+1=3となるので，関数FETCH_STRにより
S_TEMP = "#"がセットされる．

(1-3): 関数STR_CMPで残りの入力ストリング"#"と
S_TEMPが比較され，bc#∈K'が判定される．

4．　ダブル配列の更新

4．1　キーの追加

ダブル配列の初期状況は，CHECK[1]とBASE[1]が共に1であって，最大インデックスCHECK[1]を越える要素の値は0とする．このとき，追加アルゴリズム２はアルゴリズム１の行(1-1,1-3)のreturn(FALSE)を次のように変更することで得られる．

a) 行(1-1)のretun(FALSE)の変更:

```
begin
   A_INSERT(s, a_h a_{h+1} ... a_n a_{n+1});
   return(FALSE)
end
```

b) 行(1-3)のreturn(FALSE)の変更:

```
begin
 /* S_TEMPをa_{h+1}a_{h+2} ... a_{h+r}b_1 ... b_m，残りの入
力ストリングをa_{h+1}a_{h+2} ... a_{h+r}a_{h+r+1} ... a_n a_{n+1}
として，b_1≠a_{h+r+1},0≦r≦n-h+1,0≦m と仮定する*/
   B_INSERT(s, a_{h+1} ... a_{h+r},
            a_{h+r+1} ... a_n a_{n+1}, b_1 ... b_m);
    return(FALSE)
end;
```

アルゴリズム２で使用する主要な手続きと関数を図５～７に示す．但し，次の変数と関数を使用する．

SET_LIST(s): g(s,a)≠failなる記号aを要素とする集合を返す．

LIST, LIST1, LIST2: {#}∪Iの部分集合．

N(LIST): LISTの要素数を返す．

POS: TAILの長さに１を加えた値を保持するグローバル変数であって，初期値は1である．

current: MODIFYで変更された状態番号sに対する新しい番号を保持する（詳細は以下で説明される）．

X_CHECK(LIST): c∈LISTなる全てのc(nullは除く)がCHECK[q+c]=0を満足する最小のインデックスqを返す．

SET_STR(p,y): ストリングyをTAILのインデックスpの位置から連鎖し，pがPOSと等しければPOSにyの長さを加えた値を，そうでなければPOSの値を返す．

STR_LEN(z): ストリングzの長さを返す．

```
procedure A_INSERT(s,b_1 b_2 ... b_m );
begin
(a-1) t:=BASE[s]+b_1 ;
(a-2) if CHECK[t]≠0 then
      begin
(a-3)    LIST1:=SET_LIST(s);
(a-4)    LIST2:=SET_LIST(CHECK[t]);
(a-5)    if(N(LIST1)+1<N(LIST2)) then
(a-6)      s:=MODIFY(s,s,b_1 ,LIST1)
         else
(a-7)    s:=MODIFY(s,CHECK[t],null,LIST2);
      end;
(a-8) INS_STR(s,b_2 ... b_m ,POS);
 end;
```

図５　手続きA_INSERT

```
function MODIFY(current,s,a,LIST);
 begin
(m-1) old_base:=BASE[s];
(m-2) BASE[s]:=X_CHECK(LIST∪{a});
      for each c in LIST do
      begin
(m-3)   t:=old_base+c; t':=BASE[s]+c;
(m-4)   CHECK[t']:=s;  BASE[t']:=BASE[t];
(m-5)   if BASE[t]>0 then
        begin
(m-6)     for each q such that CHECK[q]=t do
(m-7)       CHECK[q]:=t';
(m-8)       if(t=current) current:=t';
        end;
(m-9)     BASE[t]:=0;  CHECK[t]:=0
      end
      return(current)
end;

procedure INS_STR(s,e_1 e_2 ... e_q ,d_pos);
begin
(s-1) t:=BASE[s]+e_1 ;
(s-2) CHECK[t]:=s; BASE[t]:=-d_pos;
(s-3) POS:=SET_STR(d_pos,e_2 e_3 ... e_q $);
end;
```

図６　関数MODIFYと手続きINS_STR

```
   procedure B_INSERT(s, x, y, z);
   /*x=b1b2 ... bk; y=c1c2 ... ch;
     z=d1d2 ... du;*/
begin
(b-1) old_pos := -BASE[s];
      for i:=0 to STR_LEN(b1b2 ... bK) do
      begin
(b-2)   BASE[s]:=X_CHECK({bi});
(b-3)   CHECK[BASE[s]+bi]:=s;
(b-4)   s:=BASE[s]+bi
      end;
(b-5) BASE[s]:=X_CHECK({c1,d1});
(b-6) INS_STR(s,d1d2 ... du, old_pos);
(b-7) INS_STR(s,c1c2 ... ch, POS)
   end;
```

図７　手続きB_INSERT.

手続きA_INSERTは，ダブル配列上でキーがミスマッチしたときにキーを追加し，もしBASE[s]の変更が必要ならば，次の優先順位でMODIFYを呼ぶ．

N(LIST1)+1（状態s_rからの遷移数と新しい遷移$g(s_r,b_1)$の和）とN(LIST2)（CHECK[t]から出る遷移数）を比較して，少ない方の状態番号のBASEの値を変更する（詳細は定理１の証明と例３を参照）．

手続きINS_STRは，セパレート状態に至る遷移を定義し，シングルストリングをTAILに格納する．また，手続きB_INSERTは，キーがTAIL上でミスマッチしたときにキーを追加する．第2引き数は，残りの入力ストリングxyとS_TEMP(xzとする)で部分マッチした接頭辞x；第3,4引き数はxy,xzからこの接頭辞xを除いたストリングy,zをそれぞれ表す．

ダブル配列の大きさCHECK[1]は関数X_CHECKで変更されるものとし，以後条件Ｂの議論は省略する．

［定理１］　アルゴリズム２は，x∉Kなる入力xをダブル配列に追加し，得られた配列は条件Ａを満足する．

（証明）　各手続きの正当性を証明する．

1)　手続きA_INSERT.

行(a-2)の条件式が偽の場合，行(a-8)で呼ばれるINS_STRの行(s-1,s-2)は遷移$g(s,b_1)$をダブル配列に定義し，行(s-3)はシングルストリングをTAILに格納する．しかも，tは必ずセパレート状態番号となるので，条件Ａは明らかに保存される．また，行(a-2)が真で，行(a-5)の比較により行(a-6)のMODIFYが呼ばれたとき，条件(A-1)の成立は次の３点ついて証明すればよい．

a)　新しいBASE[s]による遷移の定義．

行(m-2)のX_CHECKが決定する新しいBASE[s]は，b∈LIST∪{a}なる全てのbに対してCHECK[BASE[s]+b]=0を満足し，行(m-4)は$g(s,c)$，c∈LISTなる遷移を定義する．また，行(m-8)はcurrent(=s)を変更しないので，行(a-8)で呼ばれるINS_STRの行(s-1,s-2)は新しい遷移$g(s,b_1)$を正しくダブル配列上に定義する．

b)　他の遷移への影響．

BASE[s]の変更に伴い行(m-3)の番号tは新しい番号t'変化するが，行(m-4)がBASE[t]をBASE[t']にコピーし，行(m-7)がCHECK[BASE[t]+b]=tなるCHECKの要素をt'に変更するので，矛盾は生じない．

c)　古い遷移の除去

行(m-9)により，明かである．更に，条件(A-2,A-3)の成立は，行(a-8)で呼ばれるINS_STRの行(s-2～s-3)より明かである．また，行(a-7)で呼ばれるMODIFYは，BASE[s]の代わりにBASE[CHECK[t]]の値を変更するので，番号sが変更される可能性がある．これに対して，第1引き数には番号sがcurrentとして送られ，sが変更された場合は行(m-8)がcurrentを新しい番号t'に変更するので，矛盾は生じない．

2)　手続き B_INSERT.

行(b-2～b-4)はb_1b_2 ... b_kに対する遷移を定義し，行(b-5)及び行(b-6,b-7)で呼ばれるINS_STRの行(s-2)はc_1とd_1に対する遷移を定義する．次に，行(b-6)で呼ばれるINS_STRの行(s-3)はd_2 ...d_m\$をTAIL[old_pos](old_posは行(b-1)で保持する)からオーバライトする．また，行(b-7)で呼ばれるINS_STRの行(s-3)がc_2 ... c_h\$をTAIL[POS]から新たに格納する．しかも，これらのシングルストリングの格納において，行(s-2)がBASE[t]をそれぞれ-old_posと-POSに設定する．従って，条件Ａの成立は明かである．

故に，定理は成立する．　　（証明終）．

［例３］　キーbac#,bc#,ba#,bab#の追加過程を図８に示す．ただし，最終結果は既に図４に示されている．また，行(s-3)のオーバーライトで生じたTAILの未使用要素を'?'で表す．以下にキーbc#,bab#の処理を行番号と共に簡単に記述する．

bc#では，B_INSERT(3,ε,c#,ac#)が実行される．

(b-1):old_pos=-BASE[3]=1

(b-5):BASE[3]=X_CHECK({c,a})=1

(b-6):INS_STR(3,ac#,1)

  (s-3): POS=SET_STR(1,c#$)=5

(b-7): INS_STR(3,c#,5)

  (s-3): POS=SET_STR(5,#$)=7

bab#ではA_INSERT(2,b#)が実行される．

(a-1,a-2): t=BASE[2]+b=4, CHECK[4]=3≠0

(a-3): LIST1=SET_LIST(2)={c,#}

(a-4): LIST2=SET_LIST(3)={a,c}

(a-5): N(LIST1)+1=3>N(LIST2)=2

(a-7): MODIFY(2,3,null,{a,c})

  (m-1): old_base=BASE[3]=1

  (m-2): BASE[3]=X_CHECK({a,c})=6

LISTの要素aに対して，次が実行される．

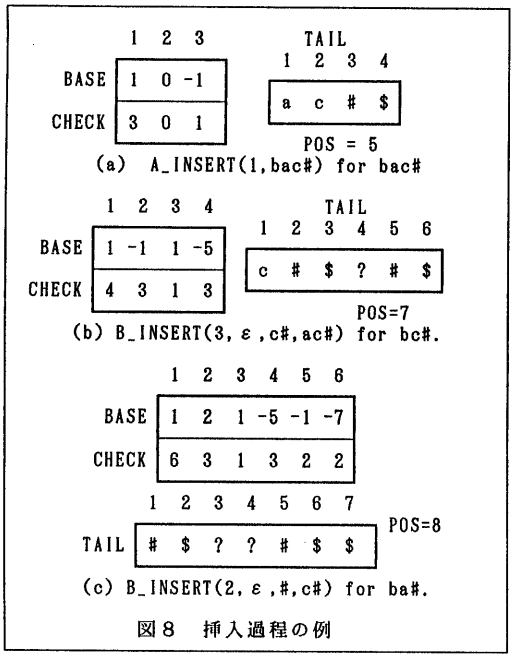

図８　挿入過程の例

(m-3): t=old_base+a=2, t'=BASE[3]+a=7

(m-4): CHECK[7]=3, BASE[7]=2

(m-5): BASE[2]=2>0

(m-7): CHECK[5]=7, CHECK[6]=7

(m-8): current=7 (現在の状態番号2の変更)

(m-9): BASE[2]=0, CHECK[2]=0

LISTの要素cに対する処理は省略するが，MODIFYは状態番号7を返し，以下が実行される．

(a-8): INS_STR(7,d#,8)

(s-3): POS=SET_STR(8,#$)=10

４．２　キーの削除

削除アルゴリズムは，アルゴリズム１の行(1-2,1-3)のreturn(TRUE)を次の様に変更することで得られる．

行(1-2,1-3)のreturn(TRUE)の変更：

```
begin
        BASE[s]:=0;    CHECK[s]:=0;
        return(TRUE)
end
```

［定理２］　アルゴリズム３は，ダブル配列からx∈Kなるキーを削除し，得られたダブル配列は条件Ａを満足する．

（証明）　負の状態番号は存在しないので，CHECK[s]=0によりセパレート状態s_rに入る$g(s_t,a)=s_r$なる遷移はダブル配列上で明らかに未定義となる．しかも，セパレート状態s_rはキーxに対して唯一であるので，定理の成立は明かである．　　（証明終）．

５．　評価

キーの総数をn，ダブル配列に定義されるべき状態数($S_M \cup S_P$の要素数)をuとするとき，$u=c_1 n$なる定数c_1は存在するが，$m=c_2 u$なる定数c_2はダブル配列の構成状況により一意に決定できないので，nとeによるmの厳密な評価は難しい．ただし，あえてnとeで評価するならばmはu・eを越えることはないので，正確な近似ではないがO(m)はO(n・e)と表せる．従って，理論的評価ではこのmを使用し，mには具体的評価を与える．なお，理論的評価はすべて最悪の場合とする．

５．１　理論的評価

以上，アルゴリズムの記述を簡単にするためにダブル配列の未使用要素の値は0として議論してきたが，X_CHECKの計算時間を決定するために，次を定義する．

ダブル配列の未使用状態番号(未使用要素のインデックス)を昇順に$r_1, r_2, \ldots, r_m$とするとき，

$$CHECK[r_i]=-r_{(i+1)};\ 1 \leq i \leq m-1$$

$$CHECK[r_m]=-1$$

なるリンクを作る．ただし，r_1は変数G_HEADで与える．このG_HEADからのリンク(G-リンクと呼ぶ)によりX_CHECKの計算時間は，O(m・e)となる．

検索の計算時間は，キーの長さkに比例しO(k)となり，また削除の計算時間(検索時間は省略)は，未使用状態番号をG-リンクへ追加する操作に依存し，O(m)となる．また，追加の計算時間(検索時間は省略)は，B_INSERTではなくA_INSERTのMODIFYに依存する．まず,X_CHECKはO(m・e)で，二つのfor文のループ回数の最大値はeとなるので，追加の計算時間は$O(m \cdot e+e^2)$となる．

ダブル配列の大きさはO(n+m)となる．また，TAILの未使用記号'?'は，キーを辞書順に追加する場合にはなくすことが可能である．即ち，B_INSERTの行(b-6)で呼ばれたINS_STRで再格納されるシングルストリングは先に格納されていたものより短くなり，常にTAILの最後尾でのみオーバーライトされるので，SET_STRの返す値を短くなった分だけPOSから引いてやればよい．

５．２　具体的評価

提案されたダブル配列によるキー検索システムは，言語Cで記述されており，1) 設計ルーチン； 2) 検索，追加，削除ルーチン； 3) 記憶管理ルーチンから成っており，それぞれ約300行である．1)は入力記号の種類と符号化(英数，片仮名等)；キー集合の分割基準(ハッシュ法，各キーの数文字分の接頭辞による部分ディジタル検索木等で定義できる)を設定する；TAILの設計(レコード情報の取り扱い等)を行う．この分割は，ダ

ブル配列要素が2バイトを越える場合に実行して，記憶領域を小さくする効果がある．また，パソコンの様に主記憶が小さくダブル配列とTAILを補助記憶からアクセスする場合は分割数を多くして，効率的な転送を実現できる．2)は，提案された検索，追加，削除のルーチンである．3)は分割されたダブル配列とTAILの記憶管理(ダブル配列とTAILの各組の格納番地と大きさをもつ表の管理)を実現する．従って，分割を行った場合は，各キーに対応するダブル配列とTAILの組の決定手順とこの記憶管理が必要となる．このシステムは，Sun Sparc Station，NEC PC-9801の種々のコンピュータ上で稼働している．

表1に示す実験結果のK_1,K_2,K_3,K_4,K_5はそれぞれPascalの指定語；COBOLの指定語；世界の主要都市名地名；英単語辞書(派生語約6万の中心語によるキー)；日本語辞書(約5万語表記の片仮名読みによるキー)であり，特にK_5は約30種類の実験結果の中から最も条件の悪い結果を与えるものとして採用した．ただし，K_4とK_5はダブル配列を五つに分割した結果であり，また記号'#'の内部コードは1に対応させてある．

表1のシングル状態数は全体の約20～70%(k_5を除けば50～70%)となり，TAILの有効性が分かる．記憶量の情報は，ダブル配列(TAILを含む)，TAIL('$'は含まない)，ソース記憶量(区切り記号を含むキー集合の大きさ)を表す．K_3以上のキー数では共通の接頭辞の効果と極端に低い未使用率(ダブル配列中の未使用要素の占有率)でダブル配列とTAILの大きさがソース記憶量の約1.1～1.2倍にしかならないことが分かる．この未使用率は，mが非常に小さい値になるという具体的評価を意味する．

表1　実験結果

各　値	K_1	K_2	K_3	K_4	K_5
各種カウント					
キーの数	35	310	1,480	23,976	32,344
キーの平均長	5.1	6.5	8.5	7.2	4.6
マルチ状態数	17	301	981	16,518	13,039
シングル状態数	74	637	5,801	35,765	11,092
全状態数	126	1,248	8,262	76,257	56,457
記憶量(K-bytes)					
ダブル配列	0.36	3.63	17.17	221	226
TAIL	0.11	0.95	7.28	60	44
ソース記憶量	0.18	2.33	14.08	196	183
未使用率(%)	17.5	8.9	0.48	0.23	0.41
時　間(ミリ秒)					
追加	31.4	30.6	29.1	36.7	35.5
検索	2.9	0.97	0.41	0.37	0.35
追加の評価					
AVE-me	1.1(3)	4.7(31)	3.6(37)	3.1(44)	19.5(255)
AVE-e	2.0(3)	2.0(7)	1.8(13)	2.5(12)	3.5(21)
AVE-e'	0.26(1)	1.5(5)	1.8(8)	1.4(10)	2.1(16)
AVE-m	1.1(3)	3.1(24	2.4(23)	2.1(32)	11.3(93)

ここで，ハッシュ法(衝突を鎖法で解決し，バケットの大きさは1とする)と2進木との記憶量の比較を行う．長さの異なるキーを検索するので，キーの代わりにキーファイルへのポインタ(pで表し，4バイトとする)を使用し，ハッシュ表の大きさをキーの総数の4分の1，その要素を2バイトと仮定すれば，バケットはポインタpと鎖用のポインタ(2バイト)を；2進木の各ノードはポインタpと二つのノードポインタ(2バイト)をもつ．この計算によりハッシュ法と2進木でキー検索を実現するための記憶量(K_3以上を対象にする)は，ソース記憶量のそれぞれ約1.7～2.2，1.8～2.4倍となるので，本手法の空間的有効性が分かる．

追加と検索時間(VAXによる値)は，主記憶上にあるキー集合を辞書順に一括して追加し，その後同じキー集合を一括検索した場合の1件当りの平均時間を表す．但し，K_1,K_2の検索時間は誤差を含む．この結果より，キー検索時間の高速性が分かる．また，追加時間も平均約35ms程度となり，十分高速であることが分かる．

追加の評価のAVE-me,AVE-e,AVE-e',AVE-mは，それぞれX_CHECKにおけるダブル配列要素の参照回数;MODIFYにおけるLISTの要素数;LISTの全要素に対して行(m-5)が真となる回数；mの平均値を表し，()内それらの最大値を表す．本実験でのeの値はK_5のみ63で，その他は96である．AVE-meよりm・eの項はK_5でも平均19.5(最大255)と決して大きい値ではない．また，AVE-e'がAVE-eよりも更に小さくなるので，e^2の項はK_5の場合でも平均132.3(最大1,008)となる．従って，これらの最大値が同時に起こったとしてもその値は1,263となり，実用的な数値であることが分かる．また，AVE-mよりmの値は，追加が連続するときには常に小さい値となることが分かる．

6．むすび

本報告では，準静的検索法と呼ばれる基準を設定し，ダブル配列によるトライ検索でこれを実現した．そして，本手法が次の点を満足することを理論的且つ具体的に示した．

1) キーの数に関係しない高速な検索が可能である．
2) 空間の使用効率が高い（キー集合の大きさ程度）．
3) 実用的な時間内でキーの追加が可能であり，しかも1)と2)の特徴を低下させない．

また，ダブル配列は如何なるキー集合に対しても構成できるので，これらの特徴を持たない完全ハッシュ法[14]，線形ハッシュ法[27]，B木法[20]；トライ検索[11,24]よりも広い適用範囲をもつキー検索法といえる．

本手法は，日英文章の解析システムに於ける辞書，高頻度英単語のフィルタリング等の約30種類のキー集合に応用されている．特に，ダブル配列は効率的ディジタル検索を実現するので，日本語解析に於ける最長一致法と英語の語彙解析におけるスペルチェックと訂正の候補[24]等の実現にも有効なものとなった．

本論文では入力記号の符号化に付いて議論しなかったが，2進符号化を導入して効率的なトライハッシング[15,19]を実現することは，最悪の追加時間を線形にするためにも興味ある課題である．

謝辞：本研究の一部は，文部省科学研究費一般研究(C)，民間企業（（株）ジャストシステム）との共同研究，（社）システム総合研究所委託研究，電気通信財団研究費などの援助を受けた．なお，本辞書検索手法は，多くの研究所に無料配布され，実用化されている．配布希望者は，770　徳島市南常三島町２－１，徳島大学工学部知能情報工学科，Tel. 0886-23-2311 内4752，Fax. 0886-55-4424に連絡されたい．

文　献

(1) A.V.Aho et al.,"Efficient string matching: An aid to bibliographic search", Commun.ACM., 18, 6, pp.333-340 (1975).

(2) A.V.Aho et al., Data Structures and Algorithms, Addison-Wesley, Reading Mass. Ch. 5 (1983).

(3) A.V.Aho et al., Compilers -Principles, Techniques, and Tools-, Addison-Wesley, Reading Mass. Ch.3 (1986).

(4) J.Aoe et al.,"A method for improving string pattern matching machines", IEEE Trans. Softw. Eng., SE-10, 1, pp.116-120 (1984)

(5) J. Aoe,"An Efficient Implementation of Static String Pattern Matching Machines," IEEE Trans. Softw. Eng., SE-15, 8, pp.1010 -1016 (1989).

(6) J. Aoe, Computer Algorithms -Key Search Strategies-, IEEE Computer Society Press. (1991).

(7) J. Aoe, "An Efficient Digital Search Algorithm by Using A Double-Array Structure, IEEE Trans. Softw. Eng., SE-15, 9, pp.1066-1077 (1989).

(8) 青江，"ﾀﾞﾌﾞﾙ配列による高速ﾃﾞｨｼﾞﾀﾙ検索ｱﾙｺﾞﾘｽﾞﾑ,"信学論(D)，J71-D，9，pp.1592-1600 (1988).

(9) 青江，"自然言語辞書の検索ーﾀﾞﾌﾞﾙ配列による高速ﾃﾞｨｼﾞﾀﾙ検索ｱﾙｺﾞﾘｽﾞﾑｰ," bit，21，6，pp.776-784 (1989)

(10) 青江,"ﾀﾞﾌﾞﾙ配列による有限状態機械の効率的ｲﾝﾌﾟﾘﾒﾝﾃｰｼｮﾝ",信学論(D)，J70-D，4，pp.653-662 (1987).

(11) A. W. Apple et al.,"The world's fastest scrabble program,"Commun. ACM, 31, 5, pp.572-578 (May,1988).

(12) F. Berman et al.:"Collections of functions for perfect hashing", SIAM J.Comput., 15, 2, pp.604-618 (1986).

(13) A.Blumer et al.,"Complete inverded filters for efficient text retrieval and analysis," Journal of ACM., 34, 3, pp.578-595, (Mar,1987).

(14) G. V. Cormack et al., "Practical perfect hashing", The Comput. J., 28, 1, pp.54-58 (1985).

(15) R. Fagin et al.,"Extensible hashing-A fast access method for dymanic files,"ACM.Trans. Database Syst., 4, 3, pp.315-344 (1979).

(16) E. Fredkin, "Trie memory," Commun. ACM, 3, 9, pp.490-500 (1960).

(17) G.H.Gonnet et al.,Handbook of Algorithms and Data Structures, Addison-Wesley, (1991).

(18) R.F.Ilson(Editor):Longman Dictionary of Contemporary English, Longman Group Ltd. (1978).

(19) W.Jonge et al.,"Two access method using compact binary trees", IEEE Trans. Softw. Eng., SE-13, 7, pp.799-809 (1987).

(20) 日高外，" 拡張B-Treeと日本語単語辞書への応用，" 信学論(D)，J67-D，4，pp.399-404 (1984).

(21) D.E.Knuth,The Art of Computer Programming, Vol.3, Sorting and Searching, Ch.6 (1973).

(22) M. Kurt, "Compressed Tries", Commun.ACM,19, 7, pp.409-415 (1976).

(23) 中嶋外，" TRIE構造とｸﾞﾗﾌｽﾀｯｸを用いた日本語形態素解析，" 情処第３９回全国大会，1F-4, pp.589-590　(1989).

(24) J.L.Peterson,Computer Programs for Spelling Correction, Lecture Notes in Comput. Sci., Springer-Verlag, N. Y.(1980).

(25) 田中，自然言語解析の基礎，産業図書 (1989).

(26) S.A.Thomas, Data Structure Techniques, Addison-Wesley, Reading Mass. Ch.3 (1980).

(27) 横山外，" 二次記憶上の大規模語彙を用いる自然言語処理システム，" 情処論，29，6，pp.570 -580 (1988)

(28) 鐘外，" Prefix-Closed B-Tree，情処自然言語研，NL73-15，(1989).