Vol. 24 No. 4 情報処理学会論文誌 Jul. 1983

パターンマッチングマシンの効率的記憶検索法†

青江順一†† 東條 隆†† 山本米雄††
島田良作†† 稲田 裕††

状態遷移図のインプリメンテーションとして，YACC (yet another compiler compiler) の語い解析部で使用されている Johnson の記憶検索法が有効である．本論文では，この Johnson の方法をさらに改善する手法を提案する．本手法では，まずパターンマッチングマシンに対する状態遷移図の特徴を見いだし，これを Johnsonの記憶検索法の検索時間と記憶量の改善に利用する．次に，状態遷移図を構成するためのキーワードの長さの情報と状態番号を表現する二つの変数とを利用して，検索アルゴリズムを能率化する．これらの改善法は理論的かつ具体的に評価され，Johnson の方法に対して次の具体的結果が得られた．(1)検索時間は約20%高速化された．(2)マシン全体（検索プログラムと状態遷移表）の記憶量は約10%軽減された．改善された検索プログラムのサイズは，Johnson のものより大きくなるが，(2)の結果はこの欠点が状態遷移表の縮小率により十分補償できることを意味している．

1. まえがき

パターンマッチングマシン (pattern matching machine) はコンパイラの語い解析，音声認識，文献検索，スペリング検査，テキスト編集など多くの分野に応用されている[1)~6)]．このマシンをインプリメントする場合，状態遷移に関する情報（この遷移関数を goto 関数と呼ぶ）を時間的かつ空間的にいかに効率よく記憶検索するかが重要な課題となってくる．goto 関数を記憶する代表的データ構造には，未定義の goto 関数も含めて一緒に蓄える行列形式(matrix form)と定義された goto 関数の情報のみを各状態に対して蓄えるリスト形式 (list form) とがあり，これらのデータ構造は検索時間と記憶量に対して対照的長所と短所をもつ．

Johnson[4)]による YACC で使用されている goto 関数の記憶検索法は Aho ら[1)]も述べているようにかなり複雑ではあるが，行列形式の時間的長所とリスト形式の空間的長所の両方をもっている．

本論文では，この Johnson の方法をパターンマッチングマシンに適用する場合に可能な改善法を提案する．本手法では，まずパターンマッチングマシンに対する状態遷移図の特徴を見いだし，これを Johnson の方法の検索時間と記憶の改善に利用する．次に，状態遷移図を構成するためのキーワードの長さの情報と状態番号を表現する二つの変数とを利用して，検索アルゴリズムを能率化する．

以下，2章ではパターンマッチングマシンと Johnson の記憶検索法について説明し，3章では Johnson の方法を改善する二つの方法を与える．4章では改善されたマシンの構成アルゴリズムを与え，5章では改善された方法に対する評価を理論的にかつ具体的に与える．

2. パターンマッチングマシン

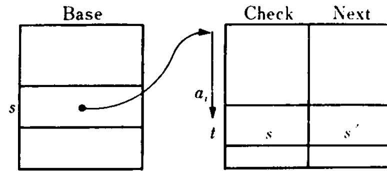
キーワード(keyword)と呼ばれるストリング(string)を要素とする有限集合をKとするとき，本論文で議論するパターンマッチングマシンはたんに入力ストリングxがキーワードの集合Kに属するか否かを判定するものとする．

Kに対するパターンマッチングマシンは，(1)状態集合S，(2)入力記号集合I，(3)状態遷移関数g，(4)判定関数f，(5)マッチングプログラムより構成されている[2)]．ここで，goto 関数である関数gは$S \times I$から$S \cup \{\mathrm{fail}\}$ への写像を定義し，関数fはSから$\{\mathrm{accept}, \mathrm{error}\}$ への写像を定義する．

状態sからs'への遷移が入力記号aに対して goto 関数により定義されていれば，$g(s, a)=s'$と書き，定義されていなければ，$g(s, a)=\mathrm{fail}$ と書く．状態は正

† An Efficient Method for Storing and Retrieving Pattern Matching Machines by JUNICHI AOE, TAKASHI TOJO, YONEO YAMAMOTO, RYOSAKU SHIMADA and HIROSHI INADA (Department of Information Science and Systems Engineering, Faculty of Engineering, Tokushima University).

†† 徳島大学工学部情報工学科



（a）　$g(s, a_i)=s'$ のデータ構造

```
begin
    s←1; i←0;
loop: i←i+1;
      t←a_i;
      t←t+base[s];
      if check[t]≠s then goto out;
      s←next[t];
      goto loop;
out:  ans←f(s)
end.
```

（b）　マッチングプログラム

図 1　Johnson の方法

Fig. 1　Johnson's method.

の整数値の番号で表されており，番号1は初期状態を表す．また，fail をデータ構造での値と密接にするために番号 0($\notin S$) で表す．したがって，$g(s, a)=0$ は $g(s, a)=$fail を意味する．

パターンマッチングマシンの入力ストリングxを

$a_1a_2\cdots a_m$；$m\geqq 1$

なる記号列とし，その右端に入力の終りを表す記号 #($\notin I$) を付けておく．また，xがKに属するか否かの結果は関数fの使用によって，$x\in K$ ならば変数 ans に accept, $x\notin K$ ならば ans に error をセットすることで得られる*.

図1に Johnson の方法による goto 関数の記憶検索法を使用したマッチングプログラムを示す．ただし，a_i は $1\leqq i\leqq m+1$ なる i を対象とし，$a_{m+1}=\#$ とする．

Johnson の方法では goto 関数を三つの1次元配列 base, check, next で表現する．この方法では，まず状態 s と入力記号 a_i に対する $g(s, a_i)$ を計算するために

$t=a_i+\text{base}[s]$

* この結果の利用はパターンマッチングマシンの応用分野により異なるので，本論文ではこの変数 ans までの情報にとどめる．

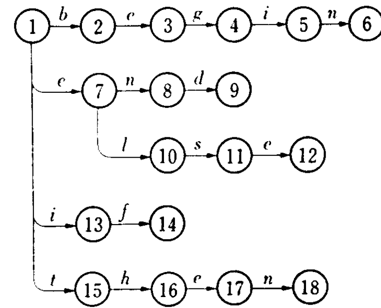


図 2　goto 関数

Fig. 2　Goto functions.

なる値 t を求める．ここで，記号 a_i は内部表現値，すなわち整数値で取り扱われている．次に，check[t] が状態番号 s と一致すれば $g(s, a_i)$ は next[t] で与えられる．そして，check[t] と s が一致しなければ $g(s, a_i)=0$ となる．

このように，Johnson の方法は入力記号の内部表現値を利用して，表 check, next のインデックス情報を効率的に得ている．

（例1）　$K=$ {begin, end, else, if, then} に対する goto 関数による状態遷移図を**図2**に示す．

図2において，状態番号1から2への b とラベル付けされた矢は $g(1, b)=2$ を示す．また，状態1は a とラベル付けされた矢をもたないので，$g(1, a)=0$ である．

表1には，図2の状態遷移図に対する Johnson の方法による表 base, check, next を示す．ただし，記号 $a, b, \cdots, z$ の内部表現値をそれぞれ $1, 2, \cdots, 26$ とする．

表 check, next 中の値0は未使用部分（goto 関数の定義のための情報が入っていない）を表すが，表 base 中の base[s]$=0$ なる状態 s は出る矢をもたない状態を意味する．すなわち，この状態 s は $\alpha\in(I\cup\{\#\})$ なるいかなる記号 α に対しても

$\text{check}[\text{base}[s]+\alpha]\neq s$

となり，goto 関数を定義しない．

さて，$g(7, n)=8$ なる goto 関数は表1より次のように表現される．

表 1　Johnson の方法による表 base, check, next

Table 1　Tables base, check and next by Johnson's method.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
表 base	−1	−3	−4	−4	−8	0	−5	6	0	−8	7	0	7	0	6	11	2	0	
表 check	1	2	3	1	4	5	7	1	7	8	10	11	13	15	16	17	0	0	1
表 next	2	3	4	7	5	6	10	13	8	9	11	12	14	16	17	18	0	0	15

base[7]$+n=-5+14=9$

check[9]$=7$

より，goto 関数は定義されており

next[9]$=8$

を得る．

3. Johnson の方法の改善

3.1 改 善 1

Johnson の方法では $t=$base[s]$+a_i$ なる値はたんに表 check, next へのインデックス情報を与えるにすぎなかったが，改善1ではこの t の値が直接遷移先の状態番号となるように表 base, check を構成する．このデータ構造において，goto 関数の確認は Johnson の方法と同じであるが，図1の遷移先の状態番号が

s←next[t]

でなく

s←t

で与えられる点に特徴がある．したがって，改善1によるデータ構造では Johnson の三つの表より表 next を取り除いたものと同様になる．

改善1の実現の保証は“パターンマッチングマシンの goto 関数による状態遷移図は一般の状態遷移図と異なり，常に初期状態を根とする木構造を成す”という特徴より得られる．すなわち，状態 s に入る遷移の矢の数を indegree(s), 状態 s から出る矢の数を outdegree(s) で表すとき，次の条件を満足する状態遷移図をT型状態遷移図と呼ぶ．

（1） 初期状態 s に対して，indegree(s)$=0$である．

（2） 初期状態でないすべての状態 s に対して，indegree(s)$=1$ である．

（3） すべての状態 s は初期状態から遷移により到達可能である．

本節の改善による表の詳細なデータ構造は，次章の表の構成アルゴリズムといっしょに説明する．

3.2 改 善 2

改善1はT型状態遷移図の特徴を利用したものであるが，改善2では従来まったく考慮されていないキーワードの情報を利用する．すなわち，キーワードの集合 K において長さが最大であるキーワードを見つけ，その長さを p とする．このとき，マッチングプログラムでは連続した正しい状態遷移数がたかだか p 回以内となるので，改善2ではこの p 回分の連続した状態遷移をループなしに実行できるプログラムを提案する．ただし，初期状態から始まる遷移回数が奇数であるか

```
begin
    s←1; i←0;
    TRANSITION(1);
    TRANSITION(2);
        ⋮
    TRANSITION(p);
out1: ans←f(s);
out2: ans←f(t)
end.
```

図 3 改善されたマッチングプログラム

Fig. 3 Improved matching program.

偶数であるかにより，次の命令列を使用する．

（a） 奇数回目の状態遷移に対する命令列 ODD.

```
i←i+1;
t←a_i;
t←t+base[s];
if check[t]=s then goto out 1;
```

（b） 偶数回目の状態遷移に対する命令列 EVEN.

```
i←i+1;
s←a_i;
s←s+base[t];
if check[s]=t then goto out 2;
```

改善1と2によるマッチングプログラムを**図3**に示す．ただし，TRANSITION(j) は $1\leqq j\leqq p$ なる j が奇数のとき命令列 ODD を表し，偶数のとき命令列 EVEN を表す．

改善2の一つの特徴は，状態番号を s と t とで交互に表現することにより，改善1の s←t なる代入文を除去した点にある．そして，もう一つの特徴は連続した状態遷移をプログラムのループでなく直線的プログラムで処理し，改善1の goto loop なる文を除去した点にある．

この改善に伴い，関数 f による変数 ans への値の代入はそれぞれ s と t の状態番号について独立に実行されるが，この点はプログラムの実行能率に何ら支障を生じさせない．

4. 表の構成アルゴリズム

本章では，改善1と2によるパターンマッチングマシンを構成するために，改善1による表 base, check の構成アルゴリズムを提案する．

本アルゴリズムでは次の状態を利用する．

［定義 1］ T型状態遷移図において，次の状態を定義する．

（1） $g(s_i, a)=s_j$ なる状態 s_j が次の条件を満足するとき，状態 s_j をセパレート状態 (separate

state) と呼び，その集合を S_p で表す．

(i) outdegree(s_i)≧2 かつ outdegree(s_j)=1.

(ii) 状態 s_j から outdegree(s_k)=0 なる状態 s_k までの遷移列上に outdegree(s_l)≧2 なる状態 s_l が存在しない．

(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 (multi state) と呼ぶ．

(3) セパレート状態から outdegree(s)=0 なる状態 s までの遷移列上に存在する状態（セパレート状態は含み，状態 s は含まない）をシングル状態 (single state) と呼ぶ．

たとえば，図2でセパレート状態番号は 2, 8, 10, 13, 15；マルチ状態番号は 1, 7；シングル状態番号は 2〜5, 8, 10, 11, 13, 15〜17 である．

定義1よりシングル状態 s は outdegree(s)=1, マルチ状態 s は outdegree(s)≧1 であることがわかる．また，セパレート状態はたんにマルチ状態とシングル状態との状態遷移図上の境界を与えるだけである．

$$g(s,a)=s',\ g(s,d)=s''$$

なるマルチ状態 s を考え，a と d の内部表現値をそれぞれ 1 と 4 とする．このとき，表 base[s]=0 として表 check に情報を埋める場合，

$$\mathrm{base}[s]+a=1$$

$$\mathrm{base}[s]+b=4$$

より，check[1]=check[4]=s となり，check[2] と check[3] は未使用のままで残る．しかし，$g(s,a)=s'$ なるシングル状態 s ならば，check[t] が未使用の場合

$$\mathrm{base}[s]=t-a$$

なる base[s] を設定することで check[t]=s として未使用部分を埋めることができる．したがって，表 check の未使用部分を埋める自由度はマルチ状態よりシングル状態のほうが高いといえる．そこで，本アルゴリズムではまずマルチ状態に対する表を構成し，次にシングル状態に対する表を構成する．

セパレート状態集合 S_p の決定は容易であるので，以下 goto 関数と表の構成アルゴリズムをそれぞれアルゴリズム1，アルゴリズム2として与える．

4.1 goto 関数の構成アルゴリズム

キーワードの集合Kから goto 関数と関数fを構成するアルゴリズムを**図4**に示す．

手続 enter($a_1a_2\cdots a_m$) の **while** 文はすでに構成された goto 関数による状態遷移を辿ることを意味し，

```
Algorithm 1. Construction of the goto function.
Input. Set of keywords K={y1, y2, ..., yk}.
Output. Goto function g and decision function f.
Method. We assume that f(s) is error when state s is first
created, and g(s,a)=0 if a is undefined or if g(s,a) has not
yet been defined. The procedure enter(y) inserts into the goto
transition graph a path that spells out y.
begin
    newstate←1;
    for i←1 until k do enter(y)
end;
procedure enter(a1a2…am):
begin
    s←1; j←1;
    while g(s,aj)≠0 do
        begin s←g(s,aj); j←j+1 end;
    for p←j until m do
        begin
            newstate←newstate+1;
            g(s,ap)←newstate;
            s←newstate
        end;
    f(s)←accept
end.
```

図4 アルゴリズム1
Fig. 4 Algorithm 1.

次の **for** 文は新しく goto 関数を定義する．このアルゴリズムは Aho ら[2)]のものと基本的には同様であるので，詳細については文献2)を参照されたい．

4.2 表 base, check の構成アルゴリズム

本節のアルゴリズムでは一つのスタックを使用し，次のスタック命令，変数，関数を使用する．

push s：状態番号 s をスタックにプッシュダウンする命令．

pop：スタックのトップにある状態番号をポップアップする命令．

q：表 check のインデックス．

top：$g(s,b)$ を定義する b の中で最小の内部表現値を top(s) で表す関数（この関数の決定法は容易であるので省略する）．

図5に，表の構成アルゴリズムを示す．ただし，出力の関数 new は，状態 s の新しい状態番号を new(s) として表す関数である．また，表 base, check には最初すべて 0 の値が入っているものとし，その大きさも動的に変化するものとする．

本アルゴリズムでは，まずマルチ状態に対する表を行(1)の **for** 文で構成する．行(2)より表 check のインデックスの最も小さい未使用位置 q*を求め，そこに top(s)=a なる $g(s,a)$ の情報を蓄えるための表 base の値を行(3)で決める．そして，行(4)，(5)で

* q の初期設定が 2 となっているのは，初期状態番号 1 に対して new (1)=1 としているからである．

Algorithm 2. Construction tables base and check.
Input. Goto function *g*, function top, and S_p.
Output. Function new, and tables base and check.
Method.

```
         begin s←1; new (s)←1; push s;
            repeat
(1)           for each top state s on the stack do
                begin
                  pop; q←2;
(2)  10:          repeat q←q+1 until check[q]=0;
(3)               base[new(s)]←q−top(s);
(4)               for each b such that g(s,b)≠0 do
(5)               if check[base[new(s)]+b]≠0 then goto 10;
(6)               for each b such that g(s,b)=s' do
                  begin
(7)                 check[base[new(s)]+b]←new(s);
(8)                 new(s')←base[new(s)]+b;
(9)                 if s'∉s_p then push s'
                  end
                end
            until the stack becomes empty;
            q←2;
(10)        for each s in S_p do
              begin
(11)            for each s such that g(s,b)=s' do
                  begin
(12)                repeat q←q+1 until check[q]=0;
(13)                base[new(s)]←q−top(s);
(14)                check[q]←new(s);
(15)                s←s';
                    new(s)←q
                  end
              end
          end.
```

図 5　アルゴリズム 2
Fig. 5 Algorithm 2.

状態 s に関するすべての goto 関数の情報が表 check に重複なしに入るか否かを確認し，可能ならば行(6)〜(8)で表 check 関数 new とを決定する．不可能ならば行(2)へ帰り同様の手順を繰り返す．行(9)は後で処理されるマルチ状態のみをスタックに入れる操作を表す．

次に，セパレート状態より残りのシングル状態に対する表 base, check を行(10)，(11)で構成する．ここでは，たんに表 check の未使用部分を求め，goto 関数の情報を埋めてゆくだけでよい（行(12)〜(15)）．

（定理 1）　入力の goto 関数がＴ型状態遷移図を成すならば，アルゴリズム 2 より求まる表 base, check と関数 new は正しい goto 関数の情報を与える．

（証明）　Ｔ型状態遷移図の条件[2]を満足しない次の状態 s'' を考える．

$$g(s,a)=s'',\ g(s',b)=s''$$

$$\mathrm{top}(s)=a,\ \mathrm{top}(s')=b$$

この状態 s, s' に対してアルゴリズム 2 では

$$\begin{cases}\mathrm{check}[\mathrm{base}[\mathrm{new}(s)]+a]=\mathrm{new}(s)\\ \mathrm{new}(s'')=\mathrm{base}[\mathrm{new}(s)]+a\end{cases}$$

$$\begin{cases}\mathrm{check}[\mathrm{base}[\mathrm{new}(s')]+b]=\mathrm{new}(s')\\ \mathrm{new}(s'')=\mathrm{base}[\mathrm{new}(s')]+b\end{cases}$$

なる表 base, check と関数 new を構成する．しかし，行(2)〜(5)あるいは行(12)の操作で表 check には goto 関数の情報がけっして重複しないことが保証され，

$$\mathrm{base}[\mathrm{new}(s)]+a \neq \mathrm{base}[\mathrm{new}(s')]+b$$

が成立する．ゆえに，この場合は関数 new の 2 重定義により表 base, check の正しさは失われる．

逆に，$\mathrm{indegree}(s)\geqq 2$ なる状態 s をもたない場合は，常に関数 new は一意的に定義されるので，表 base, check は明らかに正しい goto 関数の情報を与える．

（証明終）

（例 2）　図 2 の状態遷移図にアルゴリズム 2 を適用した結果を**表 2** に示す．

check[1]，check[18]，check[19]，base[18]，base[19] は goto 関数の表現に使用されていない部分であり，とくに check[1] は初期状態番号 1 に対してのみ生じる未使用部分である．

さて，例 1 との比較として図 2 の $g(7,n)=8$ なる goto 関数を表 2 で表現すると以下のようになる．

$$\mathrm{new}[7]=5,\ \mathrm{new}[8]=6$$

より，

$$\mathrm{base}[5]+n=-8+14=6$$

$$\mathrm{check}[6]=5$$

となり，正しい表現が得られる．

表 2　改善された方法による関数 new と表
Table 2 Function new and tables by improved method.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
関数 new	1	2	3	7	8	10	5	6	11	4	12	13	9	14	20	15	16	17

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
表 base	0	−2	0	7	−8	−7	−1	−4	8	0	0	8	0	0	11	3	0	0	0	7
表 check	0	1	2	5	1	5	3	7	1	8	4	6	12	9	20	15	16	0	0	1

5. 改善法の評価

5.1 時間の評価

以上の改善は次の時間変数 (time-variables) を使用する時間式 (time-formulas) で評価される[7)~9)].

t_{index}: インデックスの与えられた一次元配列の一つの要素をレジスタにロードする時間.

t_{add}: レジスタの内容とメモリの内容を加算し，結果をレジスタに置く時間.

t_{assign}: メモリの内容をレジスタにロードする時間あるいはレジスタの内容をメモリに格納する時間.

t_{cond}: レジスタとメモリの内容を比較して，結果 CC (condition code) にセットする時間.

t_{inc}: レジスタの内容をインクリメントする時間.

t_{goto}: CC と分岐条件に従って分岐する時間.

3.2 節での状態番号 s と t および入力位置を表す i がレジスタに蓄えられているものとすれば，時間式 T は

$$T=\sum_{j\in J} b_j t_j : b_j \text{ は整数},$$

$$J=\{\text{index, add, assign, cond, inc, goto}\}$$

で表される.

まず，それぞれの方法の1回の状態遷移に対する時間式を求める．ただし，Johnson の方法による時間式を T_j, 改善1による時間式を T_1, 改善1と2の命令列 ODD と EVEN による時間式を T_2 とし，$t\leftarrow t+\mathrm{base}[s]$ は $t_{\mathrm{index}}+t_{\mathrm{add}}$ なる時間変数を対応させた.

$$T_j=4\,t_{\mathrm{index}}+t_{\mathrm{add}}+2\,t_{\mathrm{assign}}+t_{\mathrm{cond}}+t_{\mathrm{inc}}+t_{\mathrm{goto}}$$

$$T_1=3\,t_{\mathrm{index}}+t_{\mathrm{add}}+2\,t_{\mathrm{assign}}+t_{\mathrm{cond}}+t_{\mathrm{inc}}+t_{\mathrm{goto}}$$

$$T_2=3\,t_{\mathrm{index}}+t_{\mathrm{add}}+t_{\mathrm{assign}}+t_{\mathrm{cond}}+t_{\mathrm{inc}}$$

次に，$x=a_1a_2\cdots a_m a_{m+1}$, $a_1a_2\cdots a_m\in K$, $a_{m+1}=\#$ なる入力 x を探索する場合の時間式を求める．ただし，初期設定 $s\leftarrow 1$; $i\leftarrow 0$ と探索を終了するための命令列 $i\leftarrow i+1$; $t\leftarrow a_i$; $t\leftarrow t+\mathrm{base}[s]$; **if** check[t]$\neq s$ **then goto** out ; ans$\leftarrow f(s)$ に対する時間式 T_x*を次のように表す.

$$T_x=4\,t_{\mathrm{index}}+t_{\mathrm{add}}+4\,t_{\mathrm{assign}}+t_{\mathrm{cond}}+t_{\mathrm{inc}}+t_{\mathrm{goto}}.$$

また，Johnson の方法による時間式を T_J, 改善1および改善1と2による時間式をそれぞれ T_{I}, T_{II} とする.

以上より，時間式 T_J, T_{I}, T_{II} は次のようになる.

$$T_J=(m+1)\cdot T_j+T_x$$

$$T_{\mathrm{I}}=(m+1)\cdot T_1+T_x$$

$$T_{\mathrm{II}}=(m+1)\cdot T_2+T_x$$

さて，FACOM 230-38 VS[11)]の各変数の値は $k=0.96(\mu s)$ であり，$t_{\mathrm{inc}}=k$, $t_{\mathrm{index}}=t_{\mathrm{add}}=t_{\mathrm{assign}}=t_{\mathrm{goto}}=2k$, $t_{\mathrm{cond}}=4k$ となるので，上記の時間式に代入すると次のようになる.

$$T_J=(21(m+1)+25)k=21m\cdot k+46k$$

$$T_{\mathrm{I}}=(19(m+1)+25)k=19m\cdot k+44k$$

$$T_{\mathrm{II}}=(15(m+1)+23)k=15m\cdot k+38k$$

実際の処理では mk の値が大きくなるので，

$$T_{\mathrm{I}}/T_J\fallingdotseq 19/21=0.90,\quad T_{\mathrm{II}}/T_J\fallingdotseq 15/21=0.71$$

なる値が得られ，この計算機では改善1で約10%，改善1と2で約30%の検索時間の短縮が期待される.

同様に，PDA-80[12)]の次の時間変数の値より，

$k'=0.64(\mu s)$, $t_{\mathrm{inc}}=5k'$, $t_{\mathrm{index}}=t_{\mathrm{add}}=t_{\mathrm{assign}}=7k'$, $t_{\mathrm{goto}}=10k'$, $t_{\mathrm{cond}}=17k'$

時間式 T_J, T_{I}, T_{II} が

$$T_J=(81(m+1)+95)k'=81m\cdot k'+176k'$$

$$T_{\mathrm{I}}=(74(m+1)+95)k'=74m\cdot k'+169k'$$

$$T_{\mathrm{II}}=(57(m+1)+85)k'=57m\cdot k'+142k'$$

として得られる．そして，

$$T_{\mathrm{I}}/T_J\fallingdotseq 74/81=0.91,\quad T_{\mathrm{II}}/T_J\fallingdotseq 57/81=0.70$$

となり，FACOM 230-38 VS の場合と同程度の改善率が予期される.

どの計算機においても各時間変数の値の比は大差がないので，本改善法は Johnson の方法による検索時間を約30% 短縮することが理論的に期待される.

5.2 記憶量の評価

通常のT型状態遷移図において，シングル状態の数はマルチ状態の数に比べて非常に多いので*，表 check は未使用部分なしにインプリメントできる．したがって，全状態数を e とするとき，Johnson の方法における表 base, check, next の記憶量のオーダは

$$3e$$

改善1と2による表 base, check の記憶量のオーダは

$$2e$$

となり，表の記憶量に関しては改善1と2の方法により約$\frac{2}{3}$に軽減される.

5.3 シミュレーションによる評価

PASCAL[10)]の指定語と日本の代表都市名をキーワード集合 K の例として使用した．前者は，コンパイラの語い解析，後者は音声認識等による駅名の探索への適用を意図したものである．以上の結果を**表3**に与え

* この場合の命令列は s と t が入れ換わっても同じ時間式となる.

* マルチ状態とシングル状態の数に対する具体例は5.3節の表3を参照されたい.

表 3 シミュレーション結果
Table 3 Simulation results.

各 値	PASCAL指定語	日本代表都市名
キーワード数	36	46
状態数	136	239
シングル状態数	85	169
マルチ状態数	15	24
検索時間 (ms)		
T_J	84.7	485
$T_I(T_I/T_J)$	68.0(0.80)	367(0.76)
マシンの記憶量 (bytes)		
M_J	1,144	1,968
$M_I(M_I/M_J)$	1,024(0.89)	1,660(0.84)
表の記憶量		
B_J	1,084	1,908
$B_I(B_I/B_J)$	816(0.75)	1,434(0.75)
検索プログラムの記憶量		
R_J	60	60
$R_I(R_I/R_J)$	208(3.5)	226(3.8)

る．ただし，改善された方法の各値には Johnson による方法の値との比率を常に（ ）内に示す．また，記憶量においても Johnson の方法に対応する値に添字J，改善された方法による値に添字Iを付け，マシン全体の記憶量M，表の記憶量B，検索プログラムの記憶量Rを示す．PASCAL の指定語に対する検索時間は，文献13)のプログラム例に出現する指定語の入力列(4,880文字）に対する値であり，代表都市名の場合は，各都市の人口により出現率を決定した入力列(28,800文字）に対する値である．ここで，使用した計算機は FACOM 230-38 VS であり，言語は FASP である．

検索時間の結果は，5.1 節の評価に近い改善率を示している．また，記憶量では，改善された方法の検索プログラムが Johnson のものよりかなり大きくなるが，この欠点は表の縮小分により十分補償されている．

以上より，本論文で提案した Johnson の方法の改善法は適用可能な遷移図が限定されるが，全体として記憶量の増加なしに検索時間を向上させているといえる．

6. むすび

以上，パターンマッチングマシンで使用する場合の Johnson による記憶検索法を改善した．検索時間の改善率は約20%程度であるが，入力文字列を1字ごと処理するプログラムは多くの時間を要するので，この程度の改善率でも実用的有効性は十分あると思われる．

本論文では，T型状態遷移図を議論の対象としたが，本手法が一般の状態遷移図に対してどのように拡張できるかは非常に興味ある問題である．

参考文献

1) Aho, A.V. and Ullman, J.D.: *Principles of Compiler Design*, Addison-Wesley, Reading (1977).
2) Aho, A.V. and Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM.*, Vol. 18, No. 6, pp. 333-340 (1975).
3) Knuth, D.E., Morris, J.H., Jr. and Pratt, V.R.: Fast Pattern Matching in String, *SIAM J. Comput.*, Vol. 6, No. 2, pp. 323-350 (1977).
4) Johnson, S.C.; *YACC—Yet Another Compiler Compiler*, CSTR 32, Bell Laboratories, Marray Hill, N.J. (1975).
5) Peterson, J.L.: *Computer Programs for Spelling Correction, Lecture Notes in Comput. Sci.*, Springer-Verlag, New York (1980).
6) Tinsky: 表のたたみ込み, *bit*, Vol. 13, No. 2, プログラミングセミナー, pp. 52-57 (1981).
7) Cohen, J. and Roth, M.S.: Analysis of Deterministic Passing Algorithm, *Comm. ACM*, Vol. 21, No. 6, pp. 448-458 (1978).
8) Cohen, J., Situer, R. and Auty, D.: Evaluating and Improving Recursive Descent Parsers, *IEEE Trans.*, SE-5, 5, pp. 472-481 (1979).
9) 青江，山本，島田：動作パターンによる *LR*(*k*) パーサの高速化，信学論(D), J 64-D, 10, pp. 940-946 (1981).
10) Moore, L.: *Fundamental of Programming with PASCAL*, Ellis Horwood, New York (1980).
11) 富士通(株)：FACOM 230-380 SⅡVS ユーザース・マニアル (1979).
12) 日本電気(株)：μCOM-80 ユーザース・マニアル (1980).
13) 森口，小林，武市：Pascal プログラミング対話，共立出版，東京 (1981).

（昭和57年5月10日受付）
（昭和58年1月17日採録）