自然言語処理 52－4
（1985. 11. 15）

日本語テキスト用のAho-Corasick型パターン照合アルゴリズム

篠原 武
九州大学大型計算機センター

有川 節夫
九州大学理学部附属基礎情報学研究施設

はじめに

日本語テキストは英文テキストと異なり字種が多い．そのため計算機内部では通常1文字を2バイト(英字2字分)で表現している．また，多くの場合1文字1バイトのテキストの混在が許されている．本稿では，字種が多く異なる符号長の文字が混在するという特徴を持つ日本語テキストのためのパターン照合アルゴリズムについて考察する．著者らが既に開発した英文テキストのためのパターン照合アルゴリズム[1]は，Aho-Corasick[2]に基づくもので，英字1字を2つに分割する方法を用いるのでパターン照合機械のための記憶域は小さく，しかも処理時間は非常に高速である．この文字分割の方法を拡張することにより，効率よい日本語テキストのためのパターン照合アルゴリズムが実現できることを示す．

1. 日本語テキスト

まず，日本語テキストの計算機内部での表現について考察しよう．

日本語では英語などと違い多くの文字を用いる．そのため従来の1文字1バイト(7ビットまたは8ビット)の符号系では日本語テキストを表現できない．日本語テキストの計算機内部での表現は計算機メーカーによってまちまちであるが，基本的には同じで，どの場合も2バイトに漢字1文字を対応させている．また，殆どの場合従来の1バイト文字のテキストが混在できるようになっている．

日本工業規格では，計算機内部の符号系について何も規定していないが，情報交換用の符号系については規定している．そのうちのJIS C 6226は漢字符号系について規定しているが，これは従来の1バイト符号系との混在を許した符号拡張法に基づいている．以下の議論のために日本語テキストの符号系をJIS C 6226に準拠し次のように定めておく．

- 1バイトは8ビットとし，1バイト文字は$00_{(16)}$から$FF_{(16)}$までの256文字ある．
- 2バイト文字は2バイトから構成され，前半1バイト，後半1バイトはともに$21_{(16)}$から$7E_{(16)}$までである．全部で94×94ある．
- 2バイト文字への切り換えは3バイトのエスケープシーケンス1B 24 $42_{(16)}$(KIで表す)で指示する．
- 1バイト文字のうち$21_{(16)}$から$7E_{(16)}$までは1B 28 $4A_{(16)}$(KOで表す)で指示する．特に指示がない場合には1バイト文字が指示されているものとする．
- 1バイト文字のうち$21_{(16)}$から$7E_{(16)}$まで以外は常に指示されているものとする．

簡単のため，以下ではさらに次の条件を満たしているものとする．

- KI, KO以外のエスケープシーケンスは用いない．
- KI, KOは1バイト文字と2バイト文字の切り換え時にのみ用いる．

(注意) 1バイト文字のうち英数字は$21_{(16)}$から$7E_{(16)}$までに属しており，カナ文字は$A0_{(16)}$から$DF_{(16)}$に属している．

図1に日本語テキストを受理する有限オートマトンを示した．初期状態は1，受理状態は1と4である．遷移1→2→3→4はKI，4→5→6→1はKOによって行われる．1→1の遷移は$1B_{(16)}$以外の任意の1バイト文字によって行われる．4→4の遷移は1B, 21-$7E_{(16)}$以外の1バイト文字(カナ文字など)によって行われ，4→7→4の遷移は2バイト文字(漢字)によって行われる．

2. 問題点

日本語テキストのためのパターン照合における主な問題は次の2つである．

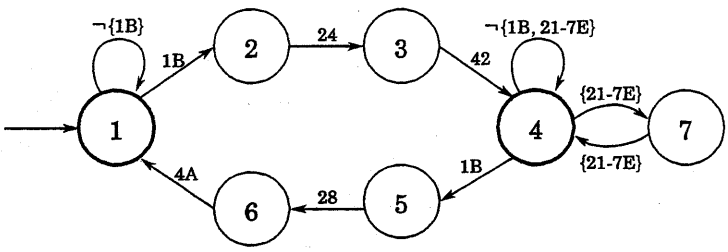

図1. 日本語テキストを受理する有限オートマトン

(1) 字種が多い. 日本語テキストは漢字など多くの文字を用いる. このことは, パターン照合アルゴリズムの実現において, 必要となる記憶域の大きさや処理速度に影響を与える. しかし, この問題は著者らが考案した文字分割の方法を用いることにより容易に解決できる. 文字分割によるパターン照合アルゴリズムの実現については第4節で簡単に紹介する.

(2) 異なる符号長の文字が混在する. 例えば…23 32 25 $50_{(16)}$…というテキストがあるとき, $23_{(16)}$が1バイト文字であるのか2バイト文字の1部であるのかは, この部分を見ただけではわからない. 従って, 日本語テキストのパターン照合においては

·1バイト文字と2バイト文字の区別,

·2バイト文字の境界合わせ

が問題となり, さらに

·KI, KOの影響を受けない1バイト文字(カナ文字など)

についても考慮しなければならないことがわかる.

3. Aho-Corasick法とBoyer-Moore法

パターン照合の問題とは, 文字列Tと文字列の集合$K=\{k_1, \cdots, k_n\}$に対し各k_iのTにおける出現位置をすべて求めるものである. ここでTをテキスト, k_iをキーワードと呼ぶことにする. パターン照合アルゴリズムの重要なものとして, Knuth-Morris-Pratt[3], Aho-Corasick[1], Boyer-Moore[4]によるものがある.

Knuth-Morris-Pratt法及びAho-Corasick法は有限オートマトンを利用したもので, 後者は前者を拡張し複数のキーワードが同時に扱えるようにしたものである. これら2つの方法はいずれもキーワードの長さに比例する時間でオートマトンを構成し, テキストの各文字を左から右へちょうど1回ずつ照合して行く.

Boyer-Moore法は1個のキーワードに対する文字列照合では最も高速なアルゴリズムとして有名である. Boyer-Moore法は, Aho-Corasick法がテキストの各文字をちょうど1回照合するのに対し, テキスト中の各文字と高々1回しか照合を行わない. Boyer-Moore法ではテキストとキーワードの照合は右から左へと行い, 不一致が検出された時点でテキストを適当な文字数だけ読み飛ばす.

日本語テキストに対して文字列照合を行う場合には, 英文テキストに対するアルゴリズムを直接適用することはできない. このことは前節で考察したことからも明らかである. Boyer-Moore法はテキストの読み飛ばしがその速さの本質であるが, 日本語テキストでは始めから順に読んで行かないと各バイトが1バイト文字なのか2バイト文字の1部なのかが区別できないので, Boyer-Moore法を日本語テキストに直接適用するのは原理的に不可能である. Aho-Corasick法も英文テキストのためのアルゴリズムを直接日本語テキストに適用することはできないが, この方法は有限オートマトンを用いているので少し手を加えるだけで日本語テキストに対しても有効となる. このことは第5節でくわしく述べることにして, ここではAho-Corasick法について英文テキストに対する基本的なものを簡単に説明しておこう.

Aho-Corasickのパターン照合アルゴリズムは, 与えられたキーワードの集合からパターンマッチングマシンと呼ばれる一種の有限オートマトンを構成する部分と, パターンマッチングマシンを与えられたテキスト上で走らせる部分から成っている. パターンマッチングマシンはgoto, failure, outputの3つの関数で構成される. キーワード{AC, BA, BB, BAA, BACD}に対するパターンマッチングマシンの例を図2に示す. 状態は円で表されており, 実線の矢印で示したのがgoto関数で, 破線の矢印で示したのがfailure関数である. 状態の右下に下線を施した文字列で示したのがoutput関数である. Goto関数は矢印の上の文字が入力されたときの状態遷移を表している.

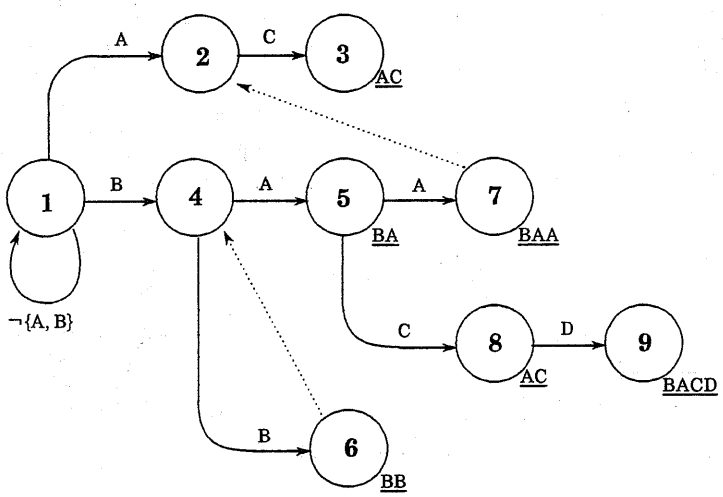

図2. Aho-Corasickのパターンマッチングマシン

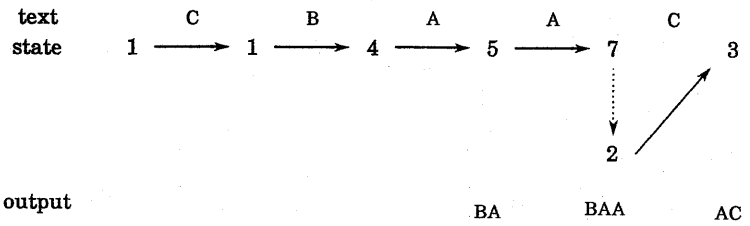

図3. パターンマッチングマシンの動作

Failure関数は入力ヘッドを動かさずに状態だけを遷移させることを表す. Output関数はその状態が出力を持つことを, すなわちキーワードの検出を表す. 図3は図2のパターンマッチングマシンをテキストCBAAC上で走らせたときの状態遷移と出力を示す.

4. 文字分割法

日本語テキストのためのパターン照合アルゴリズムにおいては, 字種が多いこと及び異なる符号長の文字が混在することが問題となるが, ここで著者らが考案し既に実現している文字分割の方法について簡単に復習しておこう.

パターンマッチングマシンはfailure関数をgoto関数に変換して, 決定性有限オートマトンにすることができる. また, goto関数は状態数×文字数のサイズのテーブルとして表現できる. 1バイト文字は256個(1バイトが7ビットの場合は128個)あるため, このテーブルサイズは小さくはない.

文字分割の方法は1文字を4ビットの文字に2分割し, テーブルサイズを1/8にするものである. 文字分割をすると状態数がほぼ2倍になるが, 処理時間は1.5倍に

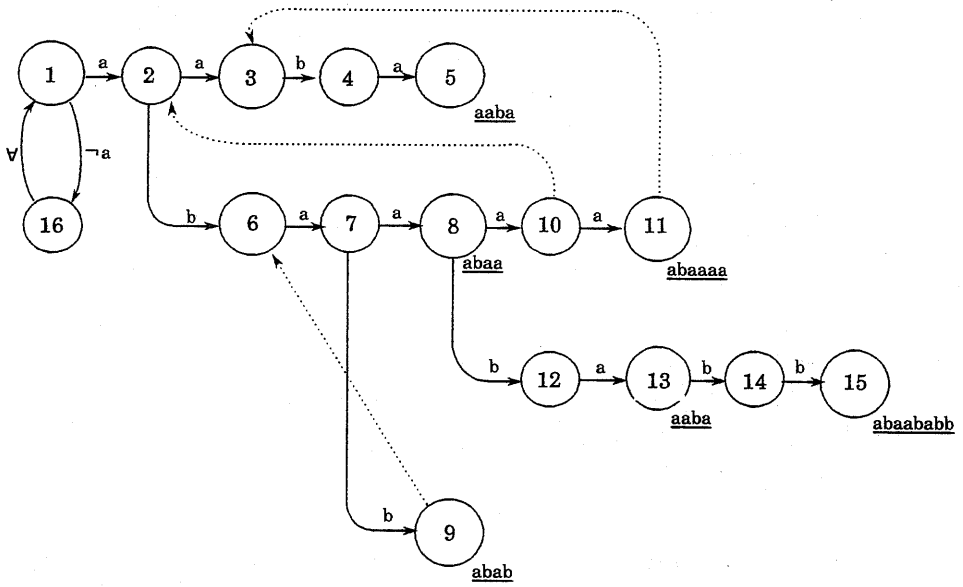

図4. 文字分割を用いたパターンマッチングマシン

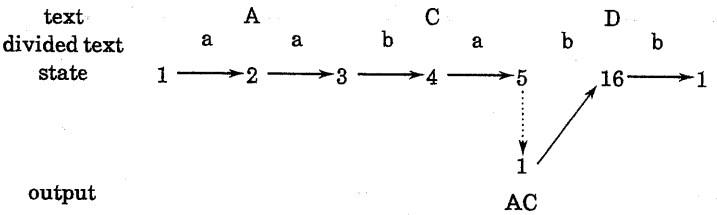

図5. 文字分割を用いたパターンマッチングマシンの動作

しかならない．しかし，単に文字を分割しただけでは誤った検出をする恐れがある．8ビット文字を大文字で，4ビット文字を小文字で表すことにし，

A = aa, B = ab, C = ba, D = bb

であるとしよう．このときテキストACD = aababbに対してパターンBB = ababを誤って検出することを防がなければならない．

この文字分割の方法は，字種の多さと境界合わせの2つの問題を解決している．これらは日本語テキストのためのパターン照合においても解決しなければならない共通の問題である．図4に文字分割によるパターンマッチングマシンの例を示す．キーワードは図2のものと同じである．そこで状態16が境界合わせのための特別なものである．大きい円から小さい円への状態遷移は2分割された前半の4ビット文字によって，小さい円から大きい円への状態遷移は後半の4ビット文字によって行われる．Failure関数による遷移は大きい円から大きい円，小さい円から小さい円へと行われる．図4においては，大きい円から状態1へのfailure関数及び小さい円から状態16へのfailure関数は省略している．

図5は図4のパターンマッチングマシンをテキストACD = aababb上で動作させたときの状態遷移と出力を示したものである．この動作例ではキーワードBB = ababの誤検出は起こっていない．ここで，状態16を付加しただけで境界合わせの問題が解決できている点に注意しよう．

5. 日本語テキストのためのパターンマッチングマシン

さて，日本語テキストのためのパターンマッチングマシンをAho-Corasickの方法に基づいて実現することについて議論しよう．

Aho-Corasickのパターンマッチングマシンはgotoおよびfailureの2つの状態遷移関数を用いるが，処理時間を短くするためにfailure関数を除去してgoto関数だけにすることができる．しかし，failure関数の除去は本質的でないので，以下ではfailure関数を用いた形のパターンマッチングマシンの構成法についてのみ議論する．また，実際の実現においては，4節で紹介したようにgoto関数のためのテーブルサイズを小さくするために4ビット文字に分割することが考えられるが，これについては触れないことにする．

図2及び図4のパターンマッチングマシンのキーワードに関係しない部分は，対象のテキストを受理する有限オートマトンと見なすことができる．日本語テキストのためのパターンマッチングマシンも同様に初期状態の部分に日本語テキストを受理する有限オートマトンを持った形で実現できる．

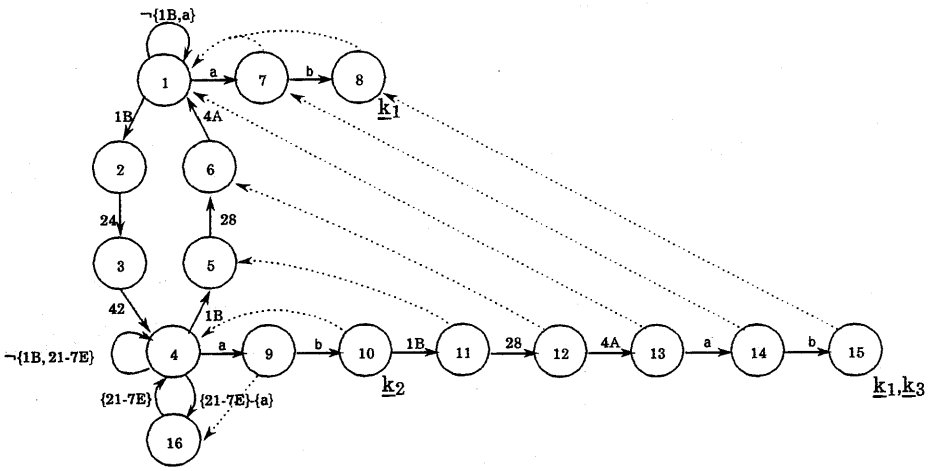

図6. 日本語テキストのためのパターンマッチングマシン

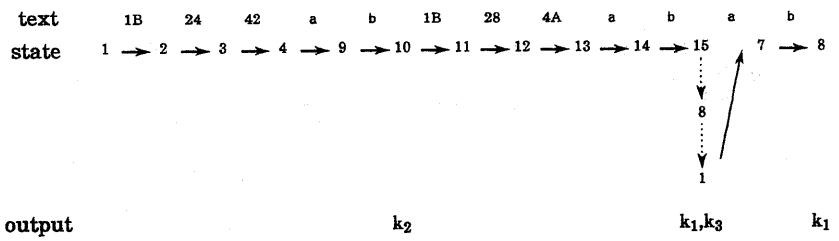

図7. 日本語テキストのためのパターンマッチングマシンの動作

以下では1バイト文字のうち21-7E$_{(16)}$に属するもの(英数字など)をa,bなどで，それ以外の1バイト文字(カナ文字など)をｱ,ｲなどで表す．また，漢字などの2バイト文字テキストはKIabcdのように表す．

5.1 1バイト文字と2バイト文字の区別

まず，1バイト文字と2バイト文字が区別できることを見るために，3つのキーワードk_1=ab, k_2=KIab, k_3=KIabKOabに対するパターンマッチングマシンを図6に示した．k_1は1バイト文字だけ，k_2は2バイト文字だけ，k_3は2バイト文字と1バイト文字が混在したキーワードである．これをテキストKIabKOabab上で動作させたときの状態遷移と出力を図7に示した．ここで，KIによって一旦2バイト文字へ切り換えられた後でも，1バイト文字のキーワードを正しく検出していることに注意しよう．

5.2 KI, KOの影響を受けない1バイト文字

次に，ｱ,ｲなどのKI, KOの影響を受けない1バイト文字を含むキーワードについて考察しなければならない．これについても上と殆ど同様に解決できるが，先頭がカナ文字で始まるキーワードに対しては，KOの状態，KIの状態に応じて2つのgoto関数を作らなければならない．

k_1=ｶﾅ, k_2=KIab, k_3=ｶﾅKIabとして，この3つのキーワードに対するパターンマッチングマシンを図8に示す．図9はこれをテキストｶﾅKIababｶﾅab上で動作させた場合の状態遷移と出力である．このとき，2バイト文字に切り換えられた状態でテキストｶﾅabが入力されるとk_3を検出しなければならない．図8のパターンマッチングマシンにおいてキーワードk_3は状態1から17への部分と4から19への部分の2箇所に対応している．状態4から19への部分は2バイト文字に切り換えられているので，k_3のKIを取り除いたものを対応させている．

5.3 日本語テキストのためのパターンマッチングマシンの構成法

さて，日本語テキストのためのパターンマッチングマシンの構成法について簡単に説明しておこう．キーワード$k_1, \cdots, k_n$が与えられたとしよう．

第1段階(初期設定) まず，図10の破線の楕円で囲んだ6つの状態からなるオートマトンを用意する．この時点では図に示した6つの矢印だけしかgoto関数は定義されていない．

第2段階(goto関数, output関数の作成) 各キーワードk_iに対して，状態1からそれまでに定義されているgoto関数に従って進めるだけ状態を遷移させる．もし

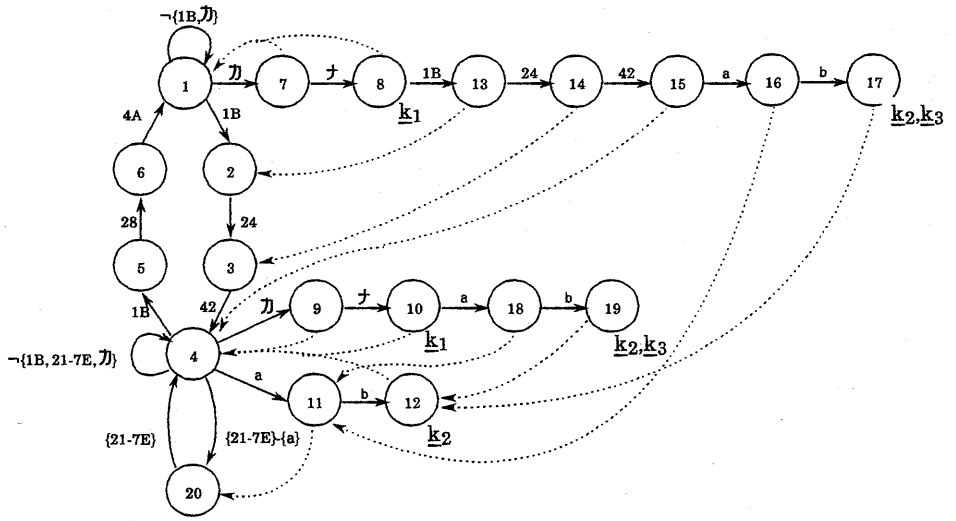

図8. カナ文字を含むキーワードのためのパターンマッチングマシン

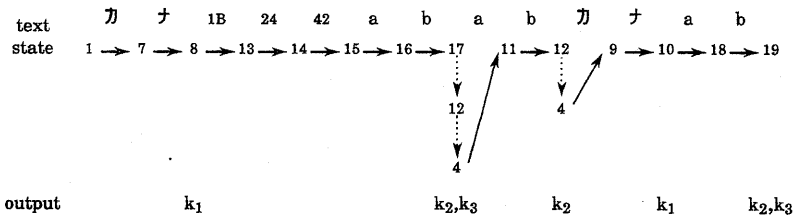

図9. カナ文字を含むキーワードのためのパターンマッチングマシンの動作

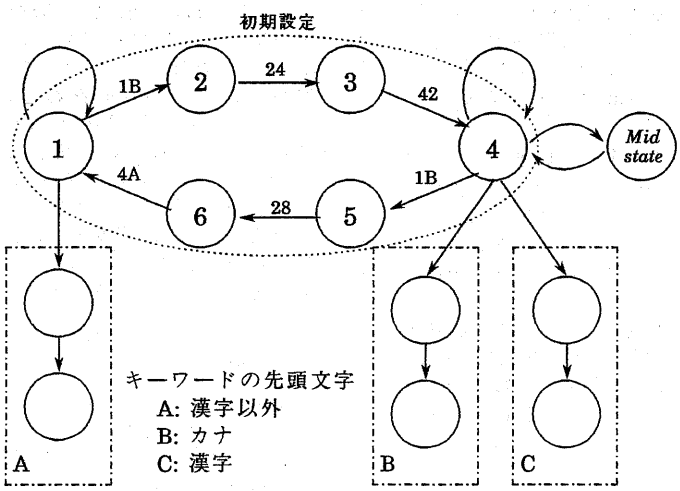

図10. 日本語テキストのためのパターンマッチングマシンの構成

キーワードの途中で遷移できなくなったら，残りの部分に対して新しくgoto関数を定義してそれに従って状態を遷移させておく．最後の状態のoutputをk_iとする．k_iがカナ文字で始まっている場合には，k_iの左から見て最初に現れるKIを除去したキーワードに対して，状態4から始めて，同様にgoto関数，output関数を作成する(図8の4→9→10→18→19の部分がこれに相当する)．

第3段階(goto関数の完成) 状態1でgoto関数が定義されていない文字に対してgoto関数を状態1自身へとする．状態4でgoto関数が定義されていない文字のうち，カナ文字に対しては状態4へ，それ以外の文字に対しては新しい状態を作りその状態へgoto関数を定義する．新しい状態のgoto関数は21-7E$_{(16)}$に対してのみ状態4へと定義し，それ以外の文字に対しては未定義としておく．ここで新しく作られた状態を*Mid-state*と呼ぶことにする(図8では状態20)．

第4段階(failure関数の作成，output関数の完成) 状態1と4を深さ0として，深さの小さい順にfailure関数を定義して行く．そのためにqueueを用いる．Queueは空に初期設定する．状態1から状態1, 2以外へのgoto関数の遷移先の各状態のfailure関数を状態1とし，各状態をqueueに入れる．状態4から状態4,5及び*Mid-state*以外へのgoto関数の各遷移先tについて，その遷移を起こす文字がカナ文字であれば状態t(図8の状態9)のfailure関数を状態4とし，そうでないならば状態t(図8の状態11)のfailure関数を*Mid-state*とする．各状態tをqueueに入れる．次に，queueが空になるまでqueueに入っている状態を取り出してfailure関数を定義していく．Queueから状態を取り出された状態をsとしよう．状態sに対してgoto関数が定義されていないならば何もしない．状態sから文字aで状態tへの遷移が定義されているとき，状態sからfailure関数で次々に遷移し，文字aに対してgoto関数が初めて定義されているときのgoto関数の遷移先uを状態tに対するfailure関数の遷移先とする．また，状態uがoutputを持つ場合にはその出力を状態tのoutputに追加する．このようなすべての状態tについてfailure関数を定義し，状態tをqueueに入れる．

6. おわりに

本稿では，日本語テキストのためのパターン照合アルゴリズムについて，日本語テキストに特有の問題点を明らかにし，その実現の方法について議論してきた．Aho-Corasickの方法に基づき，著者らがすでに開発･実現している文字分割の方法を拡張することで，1バイト文字と2バイト文字が混在する日本語テキストをそのまま1バイト単位で照合するアルゴリズムを構成することができた．最後に，その他のパターン照合問題においても，本稿で用いた技法が適用できることを簡単に補足しておく．

6.1 Boyer-Moore法による日本語テキストのためのパターン照合アルゴリズム

3節で議論したように，異なる符号長が混在するテキストに対して，Boyer-Mooreの方法を直接適用することはできない．しかし，一旦変換プログラムによってすべての文字の符号長を等しくすれば，Boyer-Moore法によってパターン照合を行うことができる．もちろん，変換プログラムを用いるのであるから，Aho-Corasickに基づいたアルゴリズムより処理時間は多く必要である．

異なる符号長の問題は変換プログラムを用いることで解決できるが，さらに字種の多さによる問題を解決しなければならない．Boyer-Mooreのアルゴリズムでは，テキストとキーワードの照合において不一致を検出したときに読み飛ばす文字数を，不一致を起こしたテキスト上の文字の関数によって求める．この関数はδ_2と呼ばれるもので，通常，字種の数に比例するサイズの表によって実現される．しかし，日本語

は多くの文字を用いるため関数表の大きさは数万バイトを超えてしまう.

関数δ_2は殆どの文字(キーワードに現れていない文字)に対して同じ値(キーワードの長さ)をとる.従って,関数δ_2を文字分割して2段階の表にすれば,表のサイズはそれほど大きくならない.実際,符号長を2バイトとすると,直接表にした場合には表のサイズは数万以上であるのに対し,1バイトに文字分割して2段階の表にすれば,表のサイズはそれほど大きくならない.実際,符号長を2バイトとすると,直接表にした場合には表のサイズは数万以上であるのに対し,1バイトに文字分割して2段階の表にするとそのサイズはキーワードの長さ×256程度にしかならない.しかも,表を2回引くことによる速度の低下はわずかである.

このように,文字分割の方法はBoyer-Mooreの方法に対しても有効である.しかも,この方法はδ_2のように殆どの場合に同じ値をとる関数であれば,どのようなものに対しても有効であることがわかる.従って,文字分割の方法は文字列照合ばかりでなく様々の問題に適用することができるといえる.

6.2 シフトコードを含むテキストのためのパターン照合

本稿で考察した日本語テキストは,従来の1バイト文字との混在ができるように,エスケープシーケンスを用いて1バイト文字と2バイト文字の切り換えを行うものであった.このエスケープシーケンスはシフトコードを拡張したものと考えることができる.従って,一般のシフトコードを含むテキストのためのパターン照合も本稿で用いた技法によって同様に行うことができる.

また,最近の文書作成システムなどで用いられている,字体制御記号や書式制御記号なども,一種のシフトコードと見なすことができる.こうしたシステムにおけるパターン照合も日本語テキストの場合と同様に行うことができよう.

参考文献

[1] Arikawa, S. and Shinohara, T.: "A Run-Time Efficient Realization of Aho-Corasick Pattern Matching Machines," New Generation Computing, *2* (1984) 171-186.

[2] Aho, A.V. and Corasick, M.J.: "Efficient String Matching," Commun. ACM, *18* (1975) 333-340.

[3] Knuth, D.E., Morris, J.H. and Pratt, V.R.: "Fast Pattern Matching in Strings," TRCS-74-440 (Stanford Univ., 1974).

[4] Boyer, R.S. and Moore, J.S.: "A Fast String Searching Algorithm," Commun. ACM, *20* (1977) 762-772.

Appendix

Algorithm 1

Construction of the goto function

input: Set of keywords $K=\{k_1, k_2, \cdots, k_n\}$.
output: Goto function g and partially computed output function o.
method:

```
begin
  /* stage 1 */
  g(1,1B(16)):=2; g(2,24(16)):=3; g(3,42(16)):=4
  g(4,1B(16)):=5; g(5,28(16)):=6; g(6,4A(16)):=1
  newstate:=6
  /* stage 2 */
  for i:=1 to n do
    begin
      enter(1,ki)
      if ki[1]<21(16) or ki[1]>7E(16) then
        begin
          delete_KI(ki,k')
          enter(4,k')
        end
    end
  /* stage 3 */
  for a:=0 to FF(16) do if g(1,a)=fail then g(1,a):=1
  midstate:=newstate+1
  for a:=21(16) to 7F(16) do
    begin
      if g(4,a)=fail then g(4,a):=midstate
      g(midstate,a):=4
    end
  for a:=0 to FF(16) do if g(4,a)=fail then g(4,a):=4
end

procedure enter(start,k)
begin
  state:=start; i:=1
  while g(state,k[i])≠fail and i≦|k| do
    begin
      state:=g(state,k[i])
      i:=i+1
    end
  for j:=i to |k| do
    begin
      newstate:=newstate+1
      g(state,k[j]):=newstate
      state:=newstate
    end
  o(state):={k}
end
```

```
procedure delete_KI(k,k')
begin
   for i:=1 to |k| -2 do
      if k[i]=1B(16) and k[i+1]=24(16)
         and k[i+2]=42(16) then
         begin
            k':=k[1:i-1]·k[i+3:|k|]
            return
         end
      k':=k
end
```

Algorithm 2

Construction of the failure function

input: Goto function g and output function o from Algorithm 1.
output: Failure function f and output function o.
method:

```
begin
   /* stage 4 */
   queue:=empty
   for a:= 0 to FF(16) do
      if g(1,a)≠1 and g(1,a)≠2 then
         begin
            f(g(1,a)):=1
            queue:=queue·g(1,a)
         end
   for a:=0 to FF(16) do
      if g(4,a)≠4 and g(4,a)≠5
         and g(4,a)≠midstate then
         begin
            if a<21(16) or a>7E(16) then
               f(g(4,a)):=4
            else
               f(g(4,a)):=midstate
            queue:=queue·g(4,a)
         end
   while queue≠empty do
      begin
         s:=head(queue); queue:=tail(queue)
         for a:=0 to FF(16) do
            if g(s,a)≠fail then
               begin
                  t:=g(s,a)
                  state:=f(s)
                  while g(state,a)=fail do
                     state:=f(state)
                  f(t):=g(state,a)
                  o(t):=o(t) ∪ o(f(t))
                  queue:=queue·t
               end
      end
end
```

Algorithm 3

Pattern matching machine for Japanese texts

input: A Japanese text string $X=a_1a_2...a_n$ and a pettern matching machine with goto function g, failure function f and output function o.
output: Location at which keywords occur in X.
method:

```
begin
   state:=1
   for i:=1 to n do
      begin
         while g(state,ai)=fail do state:=f(state)
         state:=g(state,ai)
         if o(state)≠empty then
            begin
               print i
               print o(state)
            end
      end
end
```