東京家政学院筑波女子大学紀要第７集　147～157ページ　　2003年

＜研究ノート＞

正規表現について
― その３　正規表現を工夫する ―

坂本　義行・江戸　浩幸

Utility Factors on Regular Expression Ⅲ

Yoshiyuki SAKAMOTO and Hiroyuki EDO

概　要

前二編において、正規表現の概要、基本的な記号およびメタ記号、正規表現エンジンに使われている非決定性有限オートマトン（NFA）と決定性有限オートマトン（DFA）の違いを学んだ。今回はその第３段階として、さらに「詳説正規表現」を読み進め、Perl、sed、grep、Tcl、Python、Expect、Emacs など、よく使われるツールに実装されている正規表現エンジンの長所を探ると同時に、その短所を避けるための工夫について検討する。

一部のバージョンによって、正規表現制御型 NFA エンジンの性質上、正規表現を微妙に変えただけで、マッチの対象や振る舞いに大きな影響がでる。DFA エンジンの場合にはまったく問題にならないことが、そこでは重要事項になる。NFA エンジンは優れた制御性を備えているため、正規表現に十分な工夫を加えることができる、ここではそのための技術について学ぶ。

１．バックトラックの意味

必要なものだけを正確に、しかも迅速にマッチさせるために、ここでは NFA エンジンにおける効率の問題を取り上げ、これをいかに利用するかを学ぶ。その鍵となるのは、バックトラックの意味を十分に理解し、可能な限りこれを避ける技法を学ぶことにある。さらに対象となる実装に最適な表現の書き方にも重要な影響を及ぼす、一般的な内部最適化の方法を学ぶ。

テストとバックトラック

特定の例の効率について解説する場合は、マッチの間にその正規表現が行った各個別のテスト回数を示す。もう一つの重要な点は、「厳密な」回数はおそらくツールによって異なる。

従来型 NFA 対 POSIX NFA

効率を分析するには、従来型 NFA なのか、POSIX NFA なのか、対象となるツールのエンジンの種類を知っていおくことが重要である。

２．興味深い例題

バックトラックと効率の問題がいかに重要であるかを示す例を見よう。

「"(\\.|[^"\\])*"」の例を挙げた。

２．１　簡単な変更－好手を先に

平均的なダブルクォート文字列には、エスケープされた文字列よりも多くの通常文字が入っているので、簡単な改善方法として、選択肢の順序を入れ換えて置く方法がある。図１はこの違いを図示したものである。下半分の矢印が減っているいるのは、１番目の選択肢のマッチ回数が増えたことを示している。つまりバックトラックの回数は減ったことになる。

効率改善を意図した変更を評価する際には、必ず押さえておかなければならない重要な点がある。

- この変更によって効果が得られるのは従来型 NFA か、POSIX NFA か、それとも両者か？
- この変更によって最大限の効果が得られるのは、テキストがマッチする時か、マッチに失敗する時か、それともいつでも効果があるのか？（解答は150ページ）

２．２　一歩進んだ方法－反復繰り返しを局所化する

図２は従来型 NFA についての例を示した

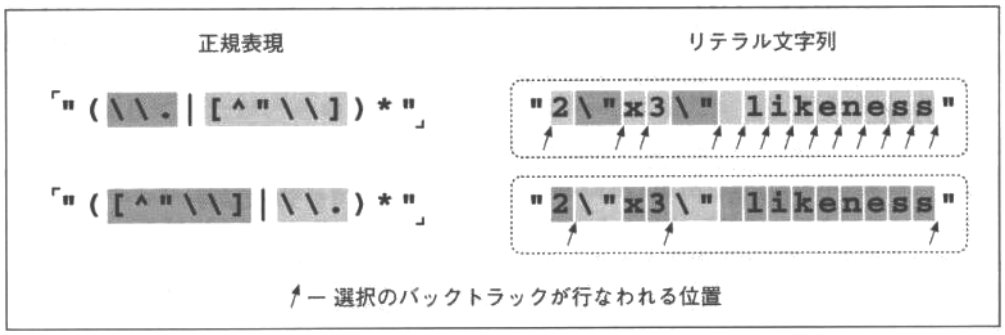

図１　選択肢の順序を変えた場合の効果（従来型 NFA）

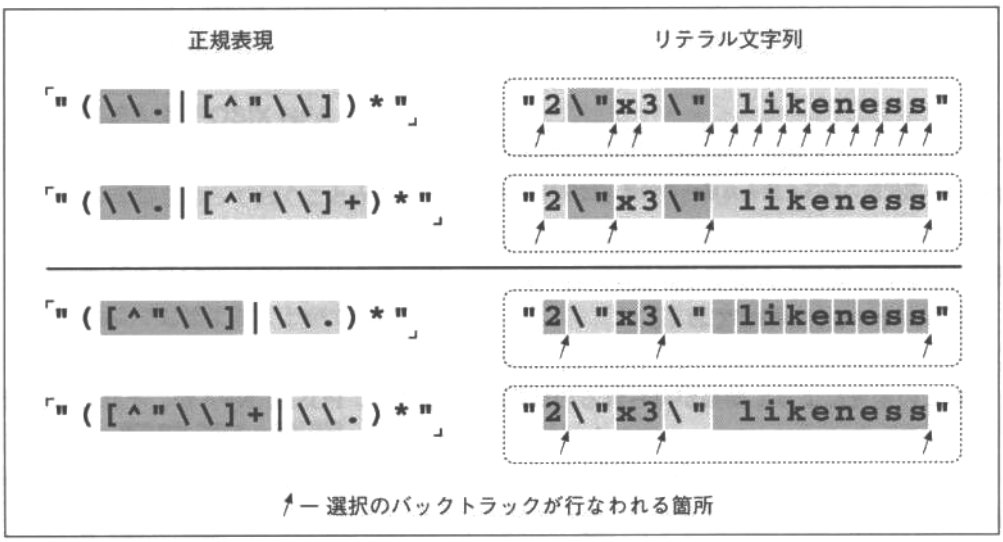

図２　プラス記号を追加した時の効果（従来型 NFA）

ものである。元の「"(\\.|[^"\\])*"」を新しい表現に置き換えると（図２の上の２つの例)、選択と関係したバックトラックとスターの反復がともに減少する。図２の下の２つの例を見ると、この変更を先ほどの順序の入れ替えと組み合わせることで、性能がさらに向上することがわかる。

表１は、演習の解答ボックスの例に似ているが、より狭い範囲を対象としたもので、スターが要求する反復の回数情報を加えてある。各ケースでは、テストとバックトラックの各回数はわずかに増加しているものの、サイクル数は激減している。

２．３　して、その正体は

これまでに POSIX NFA エンジンの威力を賞賛してきたが、その統計値は示さなかった。

プラス記号を加える前は「[^"\\]」はスターのみによって支配されており、実質的に「[^"\\]*」というパターンの取り分は限られていた。これは１文字だけにマッチしたり２文字とマッチしたりするが、可能なマッチパターン数は対象文字列の長さに直接比例していた。

実質的に「([^"\\]＋)*」と解釈されるこの新しい正規表現の場合、プラス記号とスターが文字列を分け合うパターン数は指数的に激増する。

文字列中の文字が１文字増えるごとに、可能な組み合わせが倍層する。つまり、バックトラックが行われるのだが、これが膨大な数になる。20文字による百万通り以上の組み合わせになると何秒かの時間がかかる。30文字による１兆通りを超す組み合わせでは数時間かかる。

- これらの正規表現がマッチ不成立でも速ければ DFA である。
- マッチがある場合に限って速ければ従来型 NFA である。
- 常に遅ければ POSIX NFA である。

３．バックトラックの全体像

本セクションでは、マッチおよびマッチ不成立の際に行われるバックトラックについての詳細を明らかにし、そこに見られる規則性を理解する。

手始めに、「".*"」を The name "McDonald's is said "makudonarudo" in Japanese に適用すると、そのマッチの動きは図３のように図式化できる。

正規表現は、文字列の各位置から順番にテストされるが、最初のクォートがすぐに失敗するため、テストが位置 A（図３）で始まるまでは、何も起こらない。次に、「.*」が文字列末尾にまでマッチを行うがドットが文字列末尾の無とマッチできないため、46個分のステートをマッチを行う間に蓄積してある。そこで、文字列末尾の閉じクォートをマッチテストすることになる。クォートもドットと同じように無とマッチできないので、これも失敗する。エンジンは再びバックトラックする。A から B までのマッチを行う間に蓄積されたステートが、B から C に向かって逆順にテストされる。これが12回テストされると、

表１　従来型 NFA のマッチ効率

サンプル文字列	「"([^"\\]|\\.)*"」			「"([^"\\]+|\\.)*"」		
	tests	b.t.	*-cycles	tests	b.t.	*-cycles
"makudonarudo"	16	2	13	17	3	2
"2\"x3\" likeness"	22	4	14	25	7	6
"very…99 文字…long"	111	2	108	112	3	2

簡単な変更による効果

148ページの問題の解答

どの種類のエンジンに効果があるのか？

この変更はPOSIX NFAエンジンに対しては実質的に何の効果ももたらさない。このエンジンは最終的に正規表現のすべての組み合わせをテストするので、選択肢のテスト順序は重要な問題ではない。しかし従来型NFAの場合、最初のマッチを見つけるとエンジンがそこで停止できることから、マッチにすばやく到達できるような順序で選択肢を並べて置くと効果がある。

どの結果において効果があるのか？

この変更を行うと、マッチがある時に限りマッチ速度が上がる。NFAエンジンでは、可能なあらゆるマッチの組み合わせをテストした後でなければ失敗できない（繰り返すが、POSIX NFAはいずれにしても全ての組み合わせをテストする）。そのため、実際に結果が失敗となれば、すべての組み合わせがテストされたことになるため、順序は問題とならない。

下記の表は、いくつかのケースにおけるテスト（"tests"で表記）とバックトラック（"b.t."で表記）の回数を示したものである（回数が少ないほどよい）。

サンプル文字列	従来型NFA				POSIX NFA	
	「″(\\.|[^″\\])*″」		「″([^″\\]| \\.)*″」		どちらも同じ	
	tests	b.t.	tests	b.t.	tests	b.t.
″2\″x3\″ likeness″	32	14	22	4	48	30
″makudonarudo″	28	14	16	2	40	26
″very…99文字…long″	218	109	111	2	325	216
″No \″match\″ here	124	86	124	86	124	86

ご覧のように、POSIX NFAではどちらの表現でも同じ結果になるが、従来型NFAでは、新しい表現を使った方が性能が向上する（バックトラックが減る）、事実、マッチ不成立の状況では（本表の最後の例）、どちらのエンジンもすべての可能な組み合わせを見なければならないため、いずれも同じ結果になる。

「...arudo▲" in Japa... の位置で「".*▲"」をテストせよ」というステート、すなわちCの位置にたどり着く。これはマッチ可能なので、Dに進んで全体マッチとなる。

The name "McDonald's" is said "makudonarudo" in Japanese

もしこれが従来型NFAであれば、残りの未使用のステートはそのまま破棄され、成功マッチが報告される。

3．1　さらに作業を続ける POSIX NFA

POSIX NFAの場合、先ほどのマッチは「現時点での最長マッチ」として記憶されるが、それ以上長いマッチがあるかどうかを確かめるために、残りのステートもすべてチェックされる。

3．2　マッチ不成立で行われる作業

この他に、マッチが存在しないとどうなるかを知っておかなければならない。「".*"!」

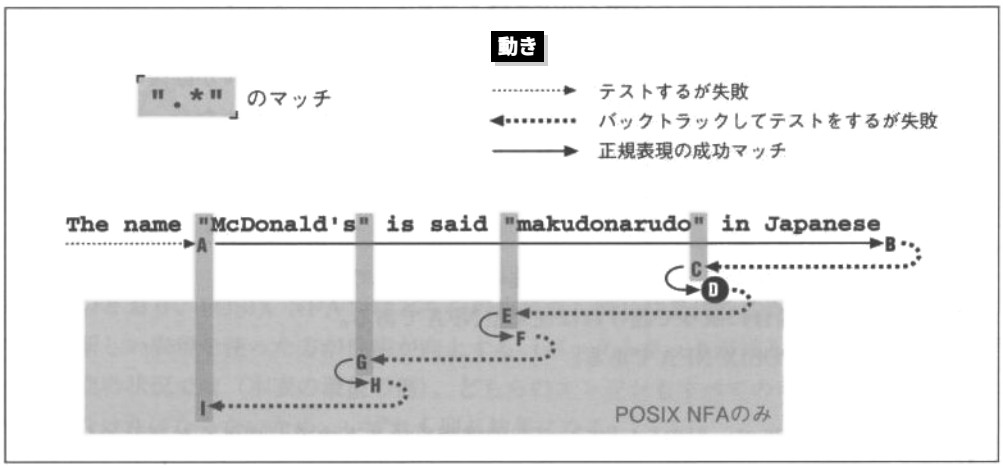

図３　「".*"」の成功マッチ

を見てみよう。図４は、この作業を図示したものである。A-I のシーケンスは図３のものと似ている。D の位置でマッチしないのが一つの違いだ（末尾の感嘆符の位置がマッチできないからである）。もう一つの違いは、図４で示されたシーケンス全体が、従来型 NFA エンジンでも POSIX NFA エンジンでも同じ結果になる。つまりすべてのものをテストすることになる。

最終的に、全体テストが失敗する。図４からも分かるように、この結果がわかるまでにかなりの作業を要する。

３．３　もっと条件を限定する

比較として、ドットを「[^"]」に置き換えてみよう。こうすればさらに絞り込まれ、処理効果も上がる。「"[^"]*"」の場合、「[^"]*」は閉じクォートを飛び越すことができないので、マッチとその後のバックトラックが大きく減る。

３．４　選択は高くつくことがある

選択はバックトラックを生む主要原因になる。単純な例として、makudonarudo のテスト文字列を使って、「u｜v｜w｜x｜y｜z」と「[uvwxyz]」がどうマッチを行うか比較してみよう。文字クラスのテストは単純なので、「[uvwxyz]」が次の位置でマッチするまで、トランスミッションシフトのバックトラックを（34回）行うだけで済む。

The name "McDonald's" is said "makudonarudo" in Japanese

しかし「u｜v｜w｜x｜y｜z」の場合、各開始位置で６回のバックトラックを行う必要があるため、同じマッチを得るには最終的に合計204回行うことになる。

３．５　過度のバックトラック

例えば、シングルクォートまたはダブルクォートで囲まれた文字列をマッチする場合、まず「'[^']*'｜"[^"]*"」という表現を使うとする。この例では，各マッチテストは単純なクォートではなく、実質的な「'｜"」で始まる。どちらの選択肢もチックしなければならない。

実は、エンジンによって最初の文字が何であるかがわかれば、この最適化を自動的に行うものがある。「先読み（lookahead）」と呼ばれる Perl の特殊機能では、「['"]」を明示的に指定することで、ちょっと先を覗いて

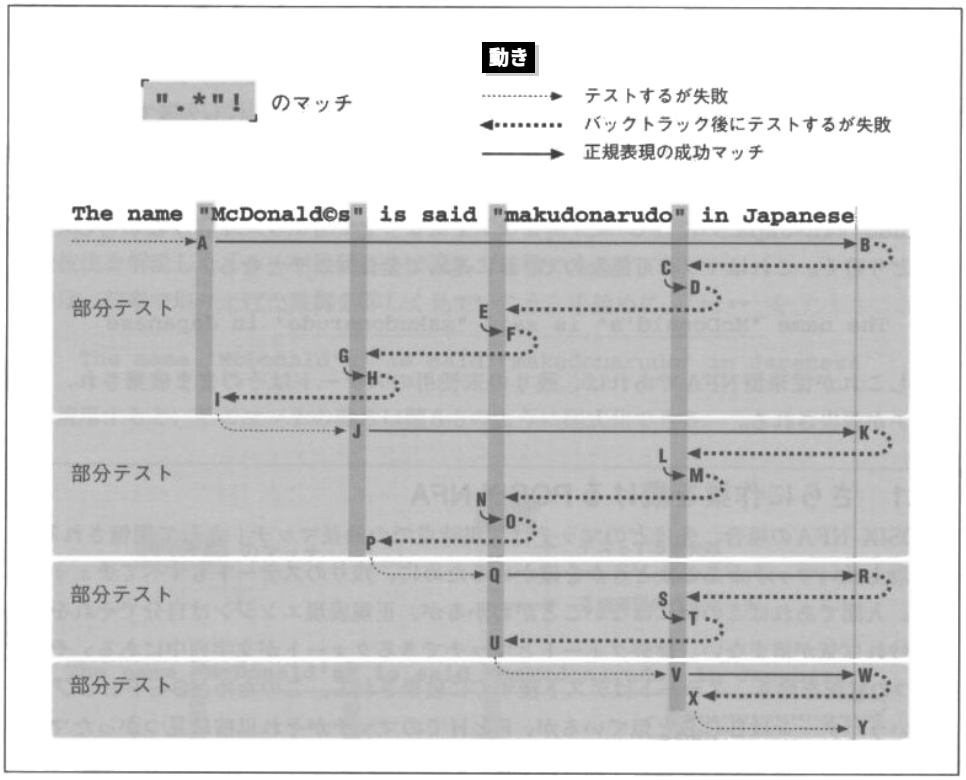

図４　「".*"!」のマッチで失敗するテスト

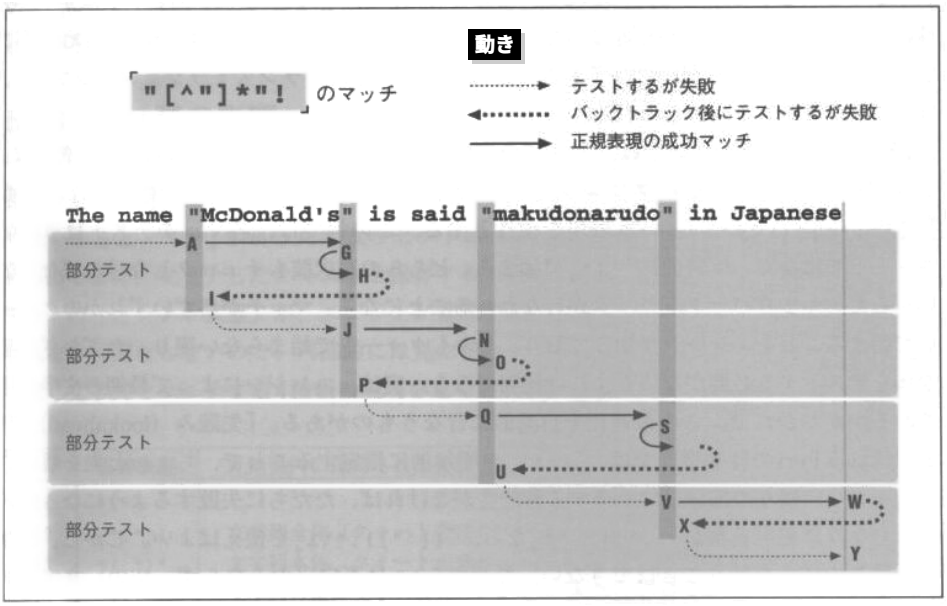

図５　「"[^"]*"!」のマッチで失敗するテスト

	正規表現	丸括弧内に保持される要素
例1	″.*″	
例2	(″.*″)	文字列全体（クォートを含む）
例3	″(.*)″	文字列全体（クォートを含まない）
例4	″(.)*″	文字列の最終文字（最後のクォートの前）

チェックし、残りの部分にマッチする可能性がなければ、ただちに失敗するようになっている。

しかし次の例を見てみよう。

「Jan｜Feb｜Mar｜Apr｜May｜Jun｜Jul｜Aug｜Sep｜Oct｜Nov｜Dec」

この場合は各テストについて12の個別チェックが必要となる。

3．6　丸括弧の影響

効率における重要な要素の一つとして、丸括弧への出入り回数と関連した格納による負担がある。例えば「"(.*)"」では、最初クォートがマッチすると、「.*」を囲む丸括弧のセットにいる。この際、正規表現エンジンは丸括弧から出るまで、マッチしたテキストを保存するために様々な準備作業を行う。内部的には見た目よりもさらに複雑になる。

同じ文字列をマッチする4種類の表現を見てみよう。

◆丸括弧とバックトラックによる性能への影響を著しく探る

まず、テストがクォート以外（マッチが成功しない位置）から始まる一般的なケースにおいて、「丸括弧の中に入る」基本的な動作によって生じる負担について考えてみよう。

例2の場合、各テストのたびに丸括弧に入り、その直後にクォートの必要性によってテストは失敗する。

テストのたびに1回ずつ丸括弧に入る負担が増えるが、複数の丸括弧にわたるバックトラックや脱出には影響しないので、最適化をしなくてもさほど問題にはならない。負担がはるかに大きいのは例4である。文字列中の文字1字ごとに丸括弧への出入りが行われるのである。

例3の負担は、例4よりはずっと低いが，少なくともマッチや部分マッチが存在するような状況では、ひょっとすると例2を多少上回るかもしれない。

Tcl（バージョン7.4）、Python（バージョン1.4bl）およびPerl（バージョン5.003）を使って、ベンチマークテストを数例行った。図6は長めの文字列に対して、何も手を加えない状態でテストをした時の時間を示したものである。

その他に予想通りだったのは、「"(.)*"」の場合に時間が急上昇する点である。Perlを含めた大半のNFAエンジンは、繰り返し制御文字が「単純」なものを支配するようなケースを最適化する。Pythonはこうした最適化は行わないので、一貫して速度が遅い。

図7のデータは、各プログラム（線で示した）ごとに例1の場合の時間を標準としてある。図7では、各線がそれぞれ独自に標準化されているため、どこか一つの正規表現について、異なる線同士を比較しても無意味である。例えばPythonの線は最も低いが、これは他の線との相対的速度とは何も関係ない。

ここから学べる教訓は、丸括弧を減らしたり、その位置を厳密に工夫することで恩恵が得られる。繰り返し制御文字を使用するなら、丸括弧は外に置くように努力するか、可能であれば削除してしまうことである。

4．内部最適化

正規表現の処理は2つの段階に分かれる。

まず正規表現を分析しながら何らかの内部形式に変換し、次には対象文字列をその内部形式に照らしてチェックする。

表現を分析する時に、よりすばやくマッチする内部形式を構築できるように、ちょっとした努力が必要である。

4．1　先頭文字の識別

「(Jan | Feb | … | Nov | Dec)？(31 | [123]0 | [012]？[1-9]）の例を見よう。

表現がテストされる各位置では、一番始めの文字でマッチが失敗することがわかるまでに、かなりのバックトラックが必要になる。

どのマッチも特定の文字（ここでは「[JFMASOND0-9]」で表現される文字）でし

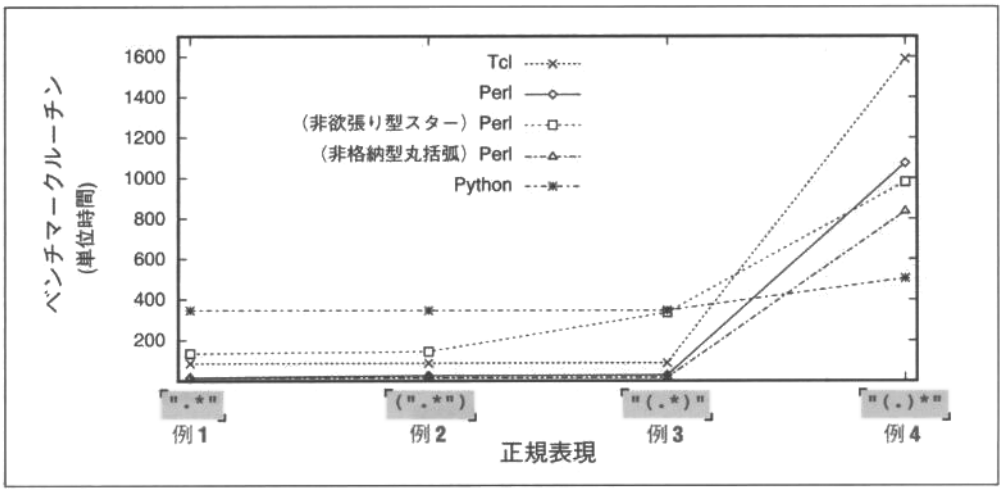

図6　丸括弧のベンチマークの数例

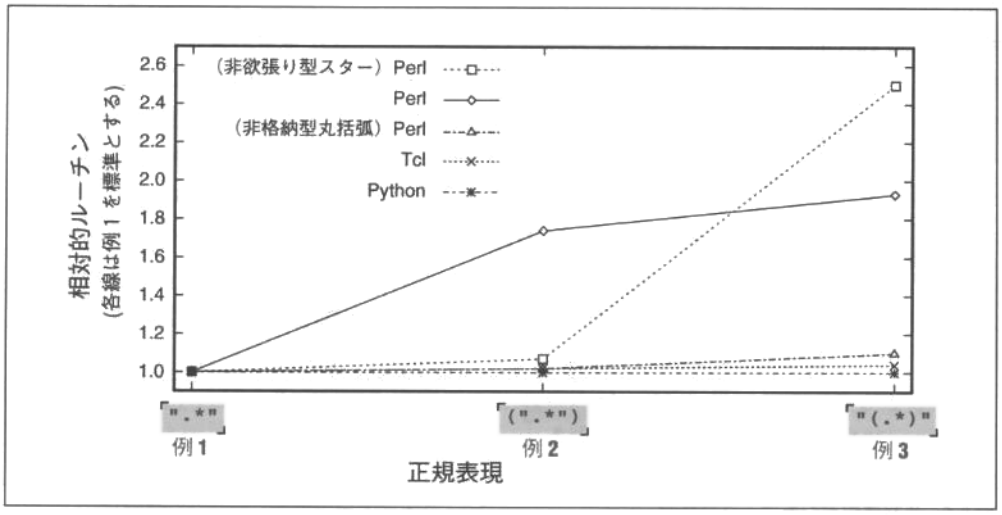

図7　変化の割合

か始まらないことが事前の分析でわかれば、トランスミッション内でその文字をスキャンし、それが見つかった時だけ制御権をエンジンに渡すようにすることが可能である。

この最適化は、マッチテストが始まる文字列内の位置を制御するトランスミッションの一部なので、どの種類のエンジンにも適用することができる。DFA では、事前にコンパイルで最初の文字の識別が万全になる。一方 NFA では、可能性のある開始文字のリストを作成するためには、さらに作業が必要となる。

４．２　固定文字列のチェック

「^Subject：_(Re：_)？(.*)」の‘ Subject：_ ’や、「".*"」のクォートのように、事前のコンパイル時の分析によって、マッチ候補の中に何らかの固定文字列が存在していることがわかれば、トランスミッションは別の技術を用いて、その固定文字列を持たない対象を即座に除外することができる。

前処理に時間をかけても、結局は時間を節約できる。一般的には「Boyer-Moore」と呼ばれるアルゴリズムが使われる。

先の最適化の場合、DFA による分析では、本来固定文字列チェックが充分サポートされているのに対し、NFA エンジンではそれほど徹底されていない。

手作業で正規表現を「th(is|at|em)」に変更すれば、NFA エンジンは、最優先固定文字列‘ th ’の後に選択が続くと考えるようになる。

４．３　単純繰り返し

リテラル文字や文字クラスのような単純な要素に、プラス記号やその仲間を使う場合、標準の NFA で生じる各ステップごとの負担の多くを省略するような最適化がしばしば行われる（この最適化はテキスト制御型の DFA エンジンには適用できない）。

「x*」、「[a-f]+」、「.？」などの場合、最適化は広く行われており、しかも大抵の場合、非常に効果がある。例えばベンチマークテストでは、「.*」の方が「(.)*」よりもかなり速い結果が出る。

図８には図７と同じデータが示されているが、「"(.)*"」の例が加えられている点が異なる。これは各線がそれぞれ正規表現１に対して標準化されている点を除けば、図６と同じである。図８から、単純繰り返しの最適化に

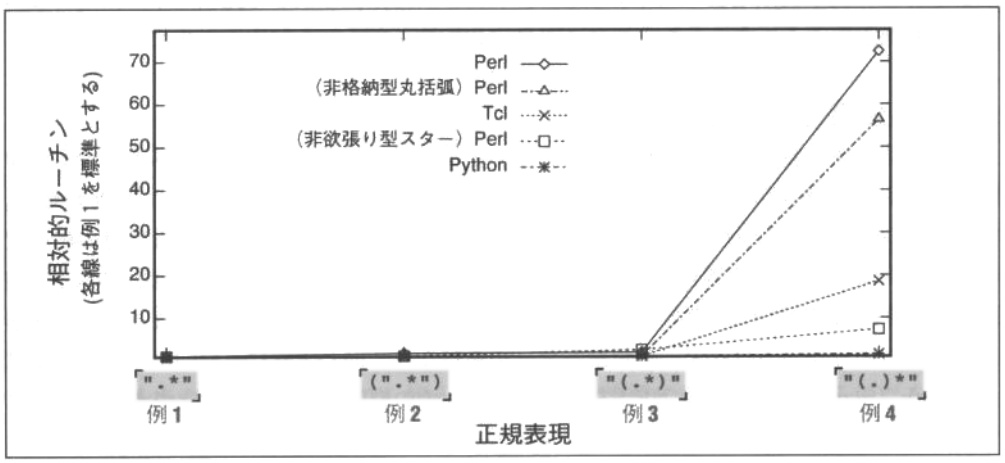

図８　変化の割合（完全データ）

ついて多くのことが分かる。

４．４　少ない繰り返しは不要

「x{3}」よりも「xxx」とする方がおそらくずっと速いだろう。{*count*} の表現は便利だが、小数の単純要素に適用する場合には、対象要素を明示的に表記することで、エンジンは出現回数を数える負担がなくなる。

４．５　文字列長認識

小規模な最適化であるが、マッチ対象がある程度の長さを持たなければならないことが分かれば、それより短い文字列は切り捨てることができる。DFA ではこの最適化をかなりうまく行うことができるが、NFA ではいい加減になることが多い。

４．６　マッチ認識

POSIX NFA が文字列末尾まで続くマッチを見つけた場合、それ以上長いマッチはあり得ないことがはっきりしているので、わざわざそれ以上検索を行う必要はなくなる。

４．７　必要性の認識

マッチの厳密な範囲が問題とならない状況で正規表現を使う場合、何らかのマッチを見つけた瞬間にエンジンが停止しても構わない。DFA および POSIX NFA ではこの最適化が最も重要になる。

４．８　文字列／行アンカー

正規表現（または各選択肢）がキャレットで始まれば、マッチは文字列先頭で始まるか、またはまったくマッチしないかのいずれかになるので、トランスミッションのシフトを省くことができる。

◆暗黙的行アンカー

関連した最適化として、「.*」で始まる正規表現が文字列先頭でマッチしなければ、それより後はどの位置においてもマッチすることはない。

４．９　コンパイルキャッシング

正規表現は、テキスト検索に用いられる前に内部形式にコンパイルされる。コンパイルには多少時間がかかるが、一度コンパイルしてしまえば、その結果は何回でも使うことができる。例として「[Tt]ubby」とマッチするファイルの行を出力する。Tcl と Perl による grep 風ルーチンを見てみよう。

どちらの例でも、正規表現はコンパイルされると、while ループが繰り返されるたびに（１回だけ）使われる。

◆関数対統合機能対オブジェクト

Tcl のマッチは regexp という通常の関数である。関数は、どのような引数が与えられるのかまったく意識していない。

Perl のマッチは演算子である。関数が呼び出され方や与えられる引数について理解できるレベルと比べれば、演算子の方が自分自身やオペランドについて多くのことを理解することができる。

人間であれば、$regex がこの例ではループを通る時には変化しないことが分かるが、Perl にはこうした高度の理解力がない。

Perl は（変数を展開した後で）新しい正規

```
－ Tcl －
while {[gets $file line] != -1} {
  if {[regexp {[Tt]ubby} $line]} {
      puts $line
  }
}
```

```
－ Perl －
while (defined($line = <FILE>)) {
    if ( $line =~ /[Tt]ubby / ) {
        print $line;
    }
}
```

```
CompiledRegex = regex.compile ("[Tt]ubby");    #正規表現をコンパイルして保存する
While 1:                                       #ファイル全体に対して
    Line = file.readline (  )                  #行を読み
    if not line: break                         #何もなければ終了する
    if (CompiledRegex.search (line) >= 0):     #コンパイルされた正規表現を行に適用し
      print line,                              #マッチがあれば出力する
```

表現と古い正規表現の間で単純な文字列比較を行ない、それが一致した場合にはコンパイルされた形式を再使用する。もし異なれば、正規表現全体が再コンパイルされる。

Tcl にまったく長所がないわけではない。Tcl は、スクリプト全体の中で一番新しく使われた５つの正規表現の内部形式をキャッシュとして保存する。

Perl では制御の一部をプログラマに任せる方法をいくつかサポートしている。Python ではプログラマが完全に制御を行えるようになっている。

この例でいえば、正規表現はループの前に一度だけコンパイルされる。そこで得られたコンパイル済み正規表現オブジェクトを次のループ内で使うことができる。

この作業はプログラマの手間を増やすが、それによって多大な制御権を得ることができる。うまく使いこなせば、効率向上につながる。

あとがき

今回は、各エンジンによって、実際の検索効率が大きく異なることに着目して、文字列の例により、どのよな検索が行われるかについて詳細な検討を加えた、とくにバックトラックの回数、効率化を図るにはどのような正規表現が有効であるかを実験により確かめた。

今後は、さらにエンジンの種類によってループ展開、C コメントの展開等について、エンジンの種類による差について検討を加えるとともに、Perl を用いて、実際の応用プログラムに適用した場合の効率的な正規表現について検証を加えていく予定である。

参考文献

１）Jeffrey E.F. Friedl著、歌代和正監訳、1999、オライリー・ジャパン発行

２）江戸浩幸・坂本義行、「電子メール自動集計システムーI」、東京家政学院筑波女子大学紀要、第４集、2000.

３）江戸浩幸・坂本義行、「電子メール自動集計システムII」、東京家政学院筑波女子大学紀要、第５集、2001.

４）坂本義行・江戸浩幸、「正規表現についてーその１どう読むかー」、東京家政学院筑波女子大学紀要、第５集、2001.

５）坂本義行・江戸浩幸、「正規表現についてーその２処理メカニズムー」、東京家政学院筑波女子大学紀要、第６集、2002.