2E1-2

状態遷移の並列比較によるNFA照合高速化

Fast NFA-based Regular Expression Matching Using Parallel Comparison

倉井 龍太郎 *1*2 Ryutaro Kurai　安田 宜仁 *1 Norihito Yasuda　湊 真一 *1*2 Shin-ichi Minato

*1JST ERATO 湊離散構造処理系プロジェクト
JST ERATO Minato Discrete Structure Manipulation System Project

*2北海道大学 大学院 情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University

Regular expression is widely used beyond the original target of text mining such as Japanese place names extraction using a large-scale pattern composed of known fragments of location names. For such applications, we are faced with the need to search for the large size of regular expression patterns whose natural constituent elements are multi-bytes. Among several methods of regular expression matching, non-deterministic finite automaton (NFA) based ones can express large regular expression compactly. Thus we focus on NFA-based matching. We introduced multi-byte symbol transitions to accelarate matching speed. However, it turns out naive introduction of multi-byte transition is not so effective, because the search time of multi-byte symbol in each state is non-negligible. To tackle this problem, we propose to use the parallel comparison, a bit-parallel technique to search symbols. Experimental results show that our method successfully decreases the matching time for practical multi-byte patterns.

1. はじめに

パターン文字列照合問題は与えられたパターン文字列と一致する部分文字列を、与えられたテキスト文字列から検索し出現位置を応答する問題である。パターン文字列照合は文書に対する検索にはじまり、テキストマイニングの前処理や、侵入検知など広く現れる情報処理における基本的な問題である。パターン文字列照合のなかでも正規表現照合は、パターンに正規表現が利用できるため複雑なパターンに対応できる重要な技術となっている。

大規模で複雑なデータ処理の需要から検索対象となるテキストのサイズは巨大化している。パターンも日本語圏においては、Wikipedia ページタイトルの抽出や人名、地域名の抽出といったマルチバイト文字での巨大な正規表現照合の利用が増加している。そのため巨大な正規表現でも高速に照合できる手法が求められている。

そこで本研究では、日本語のようなマルチバイト文字で構成される巨大な正規表現照合の高速化を目指す。手法としては、正規表現照合は Non-deterministic Finite Automata(NFA) の遷移によるものを用い、NFA の状態遷移計算を並列比較によって高速化している。

2. 正規表現照合

正規表現照合を実現する手法としては、大きく分けて 3 つの方法がある。正規表現の構文木をたどりながら照合部分文字列を発見するバックトラック法、正規表現を NFA に変換し NFA 上の遷移を行う事によって部分文字列を発見する方法、そして NFA を Deterministic Finite Automata (DFA) に変換し、DFA 上の遷移によって部分文字列を発見する方法である。バックトラック法は照合に失敗した時に構文木をだどり直すコストが大きく、パターンの一部が頻繁に出現するが、パターン全体には照合しないようなテキストに対して照合速度が遅くなる。DFA は高速に照合できるが、オートマトンの状態数が容易に指数爆発するためメモリ空間の消費が大きい [Cox 07]。NFA は正規表現パターン長に比例する状態数のオートマトンを作成することが可能であり空間効率がよい。以上の特徴から巨大な正規表現を照合する際には NFA が有力であると考えられる。ただし、NFA にはアクティブ状態の増加にともなって照合速度が低下するという問題がある [Kurai 14]。

アクティブ状態とは NFA の状態遷移において、初期状態から始まった遷移がその時点で、到達している状態である。NFA では通常アクティブ状態が複数あり、入力シンボルを受け取る度に、アクティブ状態は遷移が可能かどうかチェックされる。

正規表現から生成される NFA にはいくつかの種類があり、ε-遷移が存在するものとしない NFA がある。ε-遷移は遷移が起きる状態が増えるため、アクティブ状態を増やす原因になる。Glushkov NFA [Glushkov 61] は ε-遷移が存在せず、アクティブ状態の増加を抑えられると考えられる。このため我々は Glushkov NFA の利用に着目している。

Glushkov NFA の状態遷移には、遷移元の状態、遷移する入力シンボル、そして遷移先が格納された状態遷移テーブルを参照する必要がある。この 3 要素すべてに対して $O(1)$ でアクセスできる配列を作ると巨大なメモリ空間が必要になるので、配列で保持する要素と、探索の必要となるリストで保持する要素を組み合わせて状態遷移テーブルの参照は行われる。なかでも入力シンボルを配列で保持すると、マルチバイト文字を入力シンボルとした場合に、その値域が広いため配列が大きくなってしまう。

その対応策として、日本語を受理する NFA でも、入力シンボルはマルチバイト文字を 8bit づつ分割した値をシンボルにせざるを得ない。すると 256 種類のシンボルしか現れないため、配列は小さくなる。この方法の問題点はマルチバイト文字を分割して利用するために、本来の文字では一致が起きないような状況でも部分一致が起きることである。たとえば文字「あ」と「い」は全く一致する文字ではないが、UTF-8 で比較するとそれぞれ `E3 81 82` と `E3 81 83` というバイト列なの

連絡先: 倉井龍太郎，kurai@erato.ist.hokudai.ac.jp

で、8bit 目で一致が起きてしまう。このような部分一致はアクティブ状態の増加を引き起こし、やはり照合速度の低下につながる。

この問題に対処するために、我々はマルチバイト文字を入力シンボルとし入力シンボル数が増加するのを許容する。しかし、入力シンボルには配列を作成せず、高速な探索手法である並列比較を導入することで、状態遷移テーブル中の入力シンボルの位置を特定する。つまり、Glushkov NFA でマルチバイト文字を扱う状況において、本研究ではつぎの 2 点の改善を導入する。

- 遷移するシンボルの単位をマルチバイト文字にすることによる、アクティブ状態数の削減
- 各状態における遷移文字の探索を並列比較で行うことによる、マルチバイト文字遷移の高速化

以上の 2 つの改善が正規表現照合に与える影響を調査し、本稿ではそれらが十分に機能することを確認した。

3. 準備

3.1 Non-deterministic Finite Automata(NFA)

本稿では NFA を次の 5 個組 $A = (Q, \Sigma, \delta, I, F)$ で定義する。それぞれ、Q: 状態の有限集合, Σ: シンボルの有限集合, δ: 遷移関数, I: 初期状態の集合, F: 受理状態の集合である。遷移関数 δ は状態 $s \in Q$ とシンボル $c \in \Sigma$ を受け取って次の遷移先の集合 $n \in 2^Q$ を返す。

3.2 8bit 単位の NFA 照合

NFA 照合は正規表現によって任意に与えられる文字列パターンを NFA で保持し、検索対象となるテキストに対してパターンに一致する部分文字列を検索する問題である。

8bit 単位で遷移が行われる場合、UTF-8 でエンコードされた“(無|知) 能”という正規表現パターンは、図 1 のような NFA に変換される。

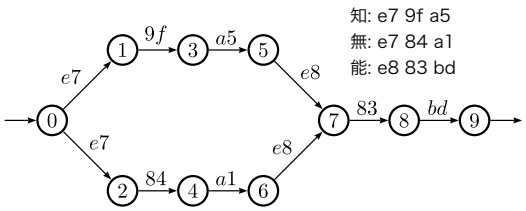

図 1: 8bit 遷移の NFA

NFA 照合では NFA への入力として、テキストからシンボルが 1 字づつ与えられる。NFA は現在アクティブになっている状態 s が入力のシンボル c での遷移 $\delta(s, c)$ を持つかチェックし、遷移がある場合は、遷移の到達先を状態をアクティブにすると動作を繰り返す。

3.3 並列比較

並列比較とは減算時のキャリービットの変化を利用して複数値の大小比較をビット並列化する手法で、Fusion Tree というデータ構造での利用が著名である [Fredman 90]。表 1 の 1 行目のように配列に n 個の数値がソートされた状態で格納されているとする。この配列に対して任意の数値 a が含まれているかを調べ、含まれている場合はそのインデックスを返したい。並列比較では、この配列の格納方法を工夫することでビット並列に探索を行っていく。表 1 の 1 行目の配列を並列比較では、表 1 の 3 行目のようなビット列 t に格納する。このビット列は利用する CPU のレジスタサイズに収まるようにする。もとのビット列 (表 1, 2 行目) の先頭に 1 ビット追加しそのビットに 1 をセットしている。そして探索したい数値の先頭に 1 ビットの 0 を追加し、n 個複製して連結したビット列 p を用意する。これは表 1 の 3 行目にあたる。この例では探索対象の数値は 5 である。この時 $t - p$ を計算すると、a よりも小さな要素は減算のキャリーによって先頭に付加されたビットが 0 になる。この計算は t, p 双方ともレジスタのサイズに収まっているので単純な整数減算で計算できる。表 1 の 4 行目を見ると比較対象の 5 より小さい値を持っていた列は先頭ビットが 0 になっていることがわかる。この付加された先頭ビットを確認していくことで、配列中のどの位置に探索したい数値 a 以上の数が格納されているかがわかる。a 以上の数の最初の位置が取得できるので、一致しているか確かめるには、実際にその位置の要素を確認する必要がある。

NFA における遷移先の探索では、シンボルを格納しソートした配列と、それぞれのシンボルで遷移する先の状態番号を保持する配列を用意する。シンボルの配列に対して並列比較を行ってシンボルが存在するインデックスを取得し、遷移先の配列に対してそのインデックスを参照することで遷移先の状態番号を得る。

表 1: 並列比較における値の変化

値 (10 進)	2	3	5	5	7
値 (3bit 2 進数)	010	011	101	101	111
フラグ付与 (t)	1010	1011	1101	1101	1111
比較対象 (p)	0101	0101	0101	0101	0101
差分 ($t - p$)	**0**101	**0**110	**1**000	**1**000	**1**010

3.4 SIMD を利用した並列比較

前節ではキャリービットを利用した並列計算について解説したが、Intel 製 CPU に搭載されている AVX 命令セットを利用すると同様の処理をより高速に行えるので本研究では本来の並列比較に変えて AVX 命令を利用した実装を利用している。AVX 命令セットは SIMD の一種で 256 ビットのレジスタを利用できる。また 256 ビットのレジスタを数ビットずつに分割し、それぞれの領域で数値比較を行うことが可能ある。

表 2 の 2 行目、3 行目のような ビット 列 を 4 ビット ずつ比較し、一致した場合にはそのブロック (4 ビット) すべてを 1 に、一致しなかった場合は全てを 0 にする命令が AVX 命令セットには存在する。実際の AVX 命令は {8,16,32,64} ビットずつの比較を行うが、ここでは説明のため 4 ビットの例を示した。

表 2: SIMD 演算における値の変化

値 (10 進)	2	3	5	5	7
値 (2 進数)	0010	0011	0101	0101	0111
比較対象	0101	0101	0101	0101	0101
比較結果	0000	0000	1111	1111	0000

4. 提案手法

4.1 マルチバイトによる遷移

NFA の遷移では 8bit で表現されるシンボルだけでなく、16bit 以上の大きさを持つ日本語の 1 字をシンボルとして遷移

を行うことも可能である。先の例で言えば、NFA は図 2 のようになる。同じパターンを受理するが大きく状態数や遷移数を減らしていて、空間効率を向上させることができる。また、図 1 の NFA では `0xe7` で始まる文字列は必ず、状態 1 と 2 をアクティブ状態にし、状態 1, 2 での状態遷移計算を発生させる。しかし、図 2 であれば、入力文字が '無 (`\u7121`)' または '知 (`\u77e5`)' である時のみ状態 1 または 2 をアクテイブにする。このようにマルチバイトのシンボルはアクティブ状態を減らすことが可能である。アクティブ状態が減少すると照合速度が向上することが知られており、このような性質は高速な NFA 照合に適している。

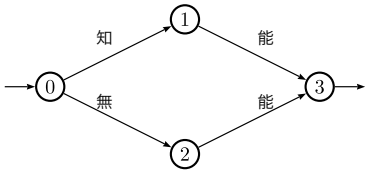

図 2: 文字遷移の NFA

そこで本研究では日本語 1 文字を状態遷移におけるシンボルとして扱う NFA を提案する。

4.2 状態を優先する状態遷移テーブル配列

Glushkov NFA の行う状態遷移計算の実現方法としては以下の 2 つの方法が考えられる。

方法 **A**: 入力シンボルで遷移できる状態をすべて取得し、現在のアクティブ状態と積集合をとる。この場合はシンボルの種類と同じサイズの配列を用意し、それぞれの要素は、対応するシンボルで遷移できる全ての状態のリストである。

方法 **B**: 状態毎に、次の入力シンボルによる遷移先を選択する。その実現のために、状態の数と同じサイズの配列を用意し、それぞれの要素には、その状態から遷移できるシンボルのリストを用意する。

方法 A ではシンボル種類数サイズの配列、方法 B では状態数サイズの配列が利用されている。巨大な正規表現では状態数も大きくなるので方法 A が選択されシンボル種類数の配列が用意されると考えられる。しかし、日本語のようなシンボル種類数が多い言語を前提とすると、シンボル種類数の配列でさえ巨大なものになってしまう。

そこで我々はシンボル種類数の配列を必要とする方法 A ではなく方法 B を利用する。また配列の要素には、遷移が可能なシンボルが連続した領域に格納されているので、その探索方法を並列比較により高速化する。

4.3 並列比較を利用した遷移先探索

状態遷移計算の高速化を行うために、本研究ではある状態からの遷移先の探索に並列比較を用いる。並列比較を可能にするために、NFA における状態遷移表をアルゴリズム 1 のような手順で生成する。

そして生成された配列 $Nexts, Positions, Length$ を利用して任意の状態 s とシンボル c に対して、次にアクティブ状態となる状態の集合をアルゴリズム 2 で求める。

アルゴリズム 2 の簡単な説明を行う。現在の注目している状態番号を s, 現在の入力シンボルを c とする。配列 $Position$ と $Length$ から、状態 s で遷移できるシンボルが配列 $Symbols$ のどの位置に格納されているかを求め、開始位置を $start$ に、利用している領域の長さを len に代入する。並列比較関数の PrallelCompare で、配列 $Symbols[start..end]$ の中で、c の始まる位置を求める。c が $Symbols$ 中に存在する場合は、配列 $Nexts$ の同じ位置に、次に遷移すべき状態の番号が格納されているので、配列 $NextActive$ にの値を格納している。

Algorithm 1 Generate Simple STT

```
pos = 0
for all s ∈ Q do
  i ← 0
  for all c such that c ∈ Σ, δ(s,c) ≠ ∅ do
    destinations ← δ(s,c)
    for all d ∈ destinations do
      Nexts[i] ← d
      i ← i + 1
    end for
    Position[s] = pos
    pos ← pos + |destinations|
    Length[s] ← |destinations|
    for i = 1 to |destinations| do
      Symbols[i] ← c
    end for
  end for
end for
```

Algorithm 2 Search Simple STT

```
Function searchNext(s,c)
start ← Positaion[s]
len ← Length[s]
index ← ParallelCompare(c, Symbols, start, end)
i ← 0
while Symbols[index] = symbol do
  NextActive[i] ← Nexts[index]
  i ← i + 1
  index ← index + 1
end while
return NextActive
EndFunction
```

5. 実験と考察

5.1 実験条件

提案手法の処理能力を計測するため、独自に正規表現照合プログラムを実装し性能比較を行った。比較対象として 8bit 単位で文字列を比較する NFA による正規表現マッチング手法 (従来手法) を実装した。また、日本語の 1 文字単位で遷移し並列比較を利用した正規表現マッチング手法 (提案手法) も実装し比較した。さらに比較のために、並列比較は行わず線形探索により次のアクティブ状態を探索する実装も用意した。正規表現パターンは表 3 のように実用的かつ大規模なパターンを 4 種類用意し、パターンの違いによる性能の変化を確認した。実験では与えたパターンと適合するすべての文字列を探索し、現れた回数をカウントした。パターンと照合するテキストは Wikipedia 日本語版の本文データから約 1GB を抽出したものを利用している。実験は 2.3 GHz Intel Core i7 を搭載した MacBook Pro で行い、OS は OS X 10.10.1 を使用している。

表 3 のパターンについて説明する。「東京都市町村名」 は東京都内に存在する、郵便番号の割り当てられている住所にマッチする正規表現である。単純にすべての住所を選言で連結するのではなく共通のプレフィックスは括りだしてまとめている。

表 3: 正規表現パターン

パターン名	パターン
東京都市町村名	(千代田区 (飯田橋|一番町|岩本町|...)|中央区 (京橋|銀座|...)|...
カタカナ	((ア|...|ン)((ア|...|ン)(ア|...|ン))((ア|...|ン)(ア|...|ン)(ア|...|ン))
英字	(A|...|Z|a|...|z|0|...|9)(A|...|Z|a|...|z|0|...|9)(A|...|Z|a|...|z|0|...|9)+
人名	(藍|相内|藍原|相羽|相庭|赤城|赤崎|赤司|...)(愛|愛之助|秋絵|朗夫|明郎|昭男|彰男|昭一|...)

「カタカナ」 は 1 から 3 文字の長さをもつカタカナにマッチする正規表現である。「英字」は半角英字と数字の 3 回以上の連続にマッチする正規表現である。「人名」は日本人の人名によく現れる姓と名をそれぞれ 1000 ずつ収集し、その組み合わせすべてに一致する正規表現である。

5.2 結果と考察

パターン毎の照合速度は表 4 のようになった。「東京都市町村名」パターンでは 70%の高速化を、「人名」では 90%以上の高速化が実現できた。このようなマルチバイト文字でしか表現できないパターンでの速度向上は当初の目標に一致している。「東京都市町村名」や「人名」で現れる NFA 上での分岐はそれぞれが 10 程度の分岐になるため、提案手法の探索による次のアクティブ状態決定が有効に機能していると考えられる。従来手法では、UTF-8 の最初 8bit から多くの分岐が発生するため、アクティブ状態数の増加が発生し速度の低下が起きている。

「英字」パターンでは従来手法と処理速度に変化が無く「カタカナ」パターンでは従来手法に劣る結果となった。どちらのパターンでも表 5 にあるように並列比較の有無で比較すると、並列比較無しでは処理時間時間が大きく悪化し、並列比較の効果により悪化の度合いが抑えられている。「英字」「カタカナ」で提案手法での高速化が出来なかった理由として次のような状況が考えられる。このパターンの中では、数回繰り返される大きな選言があり、NFA では 1 つの状態から複数の状態への分岐となる。大きな選言は 50 以上の分岐となり提案手法では分岐先の探索に大きく時間がかかる。従来手法では文字が 8bit 単位で比較され、UTF-8 における「カタカナ」の文字の最初の 8bit はすべて同じ値になる。すると、カタカナ以外の文字が誤って部分一致することが少ないため、このパターンではアクティブ状態数の増加が抑えられ、従来手法がより良い結果を出していると考えられる。

表 4: パターン毎の照合速度 (ms)

パターン名	従来手法	提案手法
東京都市町村名	63,854	25,531
カタカナ	522,495	734,557
英字	112,356	109,238
人名	850,712	52,817

表 5: 並列比較の有無による照合速度の変化 (ms)

パターン名	並列比較無し	並列比較あり
東京都市町村名	29,853	25,531
カタカナ	1,158,317	734,557
英字	139,021	109,238
人名	60,028	52,817

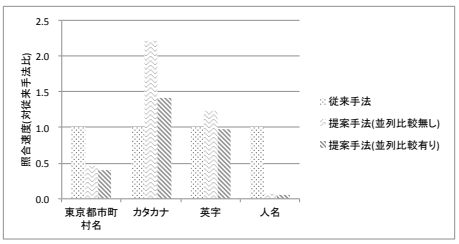

図 3: パターン毎の照合速度 (対従来手法比)

6. 結論

NFA を利用した正規表現マッチングにおいて、高速化に有効な手法の提案と評価を行った。いくつかの実用的な正規表現パターンでの検索を実際に行い、提案手法が有効であることを確認した。実験では現在入手が容易な Intel 製 CPU の SIMD 機能を利用して比較を行ったが、提案手法はそのような機材の制限を受けるものではないので、よりビット並列度の高い装置での応用や、FPGA での実装での応用が今後考えられる。遷移におけるシンボルのサイズも本稿では 16bit に広げたのみであるが、2 文字の遷移を 1 つにまとめるなどしてシンボルのサイズをより大きくすることも可能である。今後の課題としてシンボルのサイズをより大きくした時の照合速度の変化についての調査も必要である。また、並列比較を利用したアクティブ状態探索は、あらゆる形のオートマトンに応用可能であるので、有限状態トランスデューサでの利用や TRIE の探索高速化などへの利用も検討している。

参考文献

[Cox 07] Cox, R.: Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), http://swtch.com/~rsc/regexp/regexp1.html (2007)

[Fredman 90] Fredman, M. L. and Willard, D. E.: BLASTING Through the Information Theoretic Barrier with FUSION TREES, in *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pp. 1–7, New York, NY, USA (1990), ACM

[Glushkov 61] Glushkov, V. M.: The abstract theory of automata, *Russian Mathematical Surveys*, Vol. 16, No. 5, pp. 1–53 (1961)

[Kurai 14] Kurai, R., Yasuda, N., Arimura, H., Nagayama, S., and Minato, S.: Fast Regular Expression Matching Based On Dual Glushkov NFA, in *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*, pp. 3–16 (2014)