# Suffix Array を利用した正規表現検索

Regular Expression Search with Suffix Array

吉川 拓哉 †，田中久美子 ††
Takuya YOSHIKAWA, Kumiko TANAKA

† 東京大学大学院学際情報学府，†† 東京大学大学院情報理工学系研究科
†Graduate School of Interdisciplinary Information Studies, The University of Tokyo,
††Graduate School of Information Science and Technology, The University of Tokyo
†Yoshikawa.Takuya@iii.u-tokyo.ac.jp

テキストから正規表現を探し出す問題は古くから研究され応用されてきている．しかし，対象データの急速な増大と，それに平行して進むメモリの大容量化の中，新しい検索技術の必要性，そして可能性は広がり続けている．そこで本研究では，テキストの前処理とメモリの有効利用により高速な正規表現検索を構成的に行う方法を考案した．その中で，文字列の構造を再考することにより，*Suffix tree* などの現存するデータ構造の意味を別角度から捉え直している．

## 1 はじめに

テキストの中からある正規表現に該当する部分を探し出す問題は古くから研究され応用されてきている [5]．特に，理論面では計算機科学のオートマトン理論の発展と平行している．現在広く使用されている grep 等はその成果として代表的なものである．その概要は，クエリーに対応するオートマトンを作成し，テキスト上を走査することで受理状態に遷移する箇所を検索するといったところである．このような検索方法は、クエリーを前処理することによる検索の高速化技法に分類される．現在この方面の研究は，クエリーを正規表現から文字列に誤りを許容した *Approximate matching* に代表されるようなより広いクラスへ拡張する方向に進んでいる [4]．

一方，クエリーを解析する研究に対し，検索されるテキストの構造を前もって解析することで繰り返される検索のパフォーマンスを向上させるための研究も数多く報告されている．特に *Suffix tree* [6] のような文字列の構造を自然に表現したデータ構造の実用的な構築法や，その代替として使用できるよりコンパクトな *Suffix Array* [7] などに関する研究が進んできて，現実への応用も考えられるようになってきた．これらの研究は，*Suffix tree* や *Suffix Array* をより高速に構築するためのアルゴリズムの開発や，それらに代えて使用できるよりコンパクトなデータ構造の開発などが中心となって進んでいる．

このようにクエリーの前処理にはオートマトン，テキストの前処理には *Suffix tree* や *Suffix Array* とそれぞれ別のものを使用しているようだが，Blumer らの研究 [1] においてはテキストの部分文字列全てを受理するオートマトンに対応する *DAWG* を構築していて，テキストを前処理する研究にもオートマトンの理論が関わっていることが分かる．さらにアルゴリズムの解説の中ではオートマトン構成に使用される有効グラフと *Suffix tree* との間係も記述されていて，文字列の構造を理解する上でも基礎的な内容が多く含まれている。

そこで本論文では，Blumer らの論文中の文字列構造の捉え方を利用して構成的正規表現検索アルゴリズムを作成する．そのために，*Region Algebra* [2] と同様の仕方で文字列を捉え，その枠組みを正規表現検索へ拡張するというスタイルをとることにする．

## 2 関連研究

前処理付きの正規表現検索としては，Ricardo A. Baeza–Yates, Gaston H. Gonnet [3] がある．この論文では，テキストを前処理し *Patricia tree* [6] を作成しておき検索時にクエリーに相当するオートマトンをその木の上で走査させるというアルゴリズムが正規表現のうちのある部分クラスに対して平均で対数時間しかかからないことを示している．ただし，木の上のオートマトン走査というものが現実に必要とするコストやその実用法に関する考察はほとんどなされていない．よって木が必要とする空間コスト等を考えると，実用化に際してはもう一段工夫が必要となる．また、正規表現を全ての場合に対数時間で

検索することが可能かどうかという問題が未解決であることも述べられている．

## 3 文字列構造

まず準備として，検索の対象となるテキスト，つまり長大な文字列の構造をインデックスを利用して表現しておく．ただし，台となる有限アルファベット $\mathcal{A}$ には必要に応じて全順序が備わっているものと仮定する．

### 3.1 インデックス

長さ $n$ の文字列 $w = a_1a_2\cdots a_n$ はインデックスの集合 $I = \{1, 2, \ldots, n\}$ からアルファベット $\mathcal{A}$ への写像とみなすことができる．つまり各位置 $i$ に対し，$w(i) = a_i$ なる文字を対応させる写像を文字列とみる．これに対し，文字列の構造を調べるときには，これとは逆方向から文字列を捉える方が自然なこともある．つまり，各文字 $a \in \mathcal{A}$ に対し，今考えた文字列写像 $w$ による逆像 $w^{-1}(a) \subset I$ を考える．この集合は文字列 $w$ 中に存在する文字に対してはその出現位置を集めたものとなり，その他の文字に対しては空集合となる．また，インデックスの集合は，$w(I) \subset \mathcal{A}$ の大きさ，つまりアルファベット中実際にテキスト内で使用されている文字の数に分割される．この見方の利点は，検索を自然に表現できることである．この点については，$w^{-1}$ を以下のように語の集合 $\mathcal{A}^*$ から整数のペアの集合への写像と考えることで理解できる．つまり，各語 $s \in \mathcal{A}^*$ に対し，$w^{-1}(s) = \{(i, j) \mid s = a_i \cdots a_{j-1}\}$ とする．これにより，語の検索は $w^{-1}$ を計算することとして定式化される．過去の研究との関連でいえば，構造をもったテキストの抽出を可能とするための技術の一つである *Region Algebra* [2] において，この $w^{-1}$ に相当するものがインデックス関数 $\mathcal{I}$ として記述されている[1]．*Region Algebra* では，このようにインデックス関数を通して得られる整数のペアの集合上にいくつかの代数演算を定義することで，通常のキーワード検索よりも高度な検索を可能としている．ただし，それがテキストの構造とどのように関連するのかについてはあまり議論されていない．

### 3.2 部分文字列分割

後で論じる文字列の構造が Blumer らの論文でどのように論じられているかを概観する．彼らの論文では，テキスト中の部分文字列を，そのテキスト内出現における末尾の位置を集めた集合によって分類している．この分類において，各々の類を末尾位置の集合と同一視したとき，それらの集合は交わらないか包含関係にあるかのどちらかであり，全体は空文字列に対応する集合を根とする包含木になることが知られている．さらにテキストの先頭がユニークな文字であるとき，その包含木はテキストを反転した文字列の *Suffix tree* と同型である．

### 3.3 連結演算

Blumer らの論文における *Suffix tree* の捉え方[2]．が先にみた Region Algebra のインデックス関数を通した文字列の見方によりどのように表現されるのかを確認するために文字列の連結を別の形で表現しておく．つまり，二つの語 $s, t$ の連結 $st$ に対し $w^{-1}(st)$ を $w^{-1}(s)$ と $w^{-1}(t)$ で表現する[3]．

$$w^{-1}(st) = \{(i, j) \mid \exists k, (i, k) \in w^{-1}(s), (k, j) \in w^{-1}(t)\}.$$

ただし，空文字 $\varepsilon$ に対しては $w^{-1}(\varepsilon) = (I \cup (n+1)) \times (I \cup (n+1))$ としておくものとする．この式の右辺で整数のペアの集合上の要素 $w^{-1}(s)$ と $w^{-1}(t)$ の連結 $w^{-1}(s)w^{-1}(t)$ を定義する．これにより，語の連結と整数のペアの集合上の代数演算の間の対応が得られた．

### 3.4 Suffix tree 構造

文字列の構造は $w^{-1}(\mathcal{A})$ を基底として上記連結演算によって張られる代数系に集約される．つまり，*Suffix tree* 上の各点（枝上の点も考慮する）での文字列 $s$ に $w^{-1}(s)$ を対応させたとき，$s$ と $sa$ に対応する隣接する二点は連結演算 $w^{-1}(s)w^{-1}(a)$ で結ばれるのである．この対応において，ノードに対応する点とその子ノードへの枝上の点では整数ペアの集合の要素数は等しくなる．これは Blumer らの論文における同値類での分割と同等のことを連結を通して表現したからである．

## 4 正規表現検索アルゴリズム

ここまでで論じてきた文字列の捉え方に基づき，テキスト中からクエリーとして与えられた正規表現に対応する部分を探し出すアルゴリズムを与える．た

[1] *Region Algebra* においては通常，単語を一つの文字として扱い，インデックスは単語単位に割り当てられる．

[2] Blumer らの論文では，もとの文字列の反転に対する木構造に対応するので，直接対応させるためには部分文字列の先頭のインデックスにより分割しなければならない．

[3] この式は後に，正規表現検索アルゴリズムの提案において使用することになる．

だし，テキストの前処理を仮定しその結果として得る *Suffix Array* を利用する．ここで，*Suffix Array* は語 $x$ が与えられたときに，その語がテキスト中で始まる場所の集合 $S(x) = \{i \mid x = a_i a_{i+1} \cdots a_{i+|x|-1}\}$ を高速に[4]取得するために利用するだけであり，同等の役割を果たすデータ構造を代わりに用いることもできる．

### 4.1 アルゴリズム

アルゴリズムは正規表現の定義に対応し，以下のように定義される正規表現から整数のペアの集合全体への写像 $\mathcal{I}$ を任意の評価順で計算する．

- 正規表現が語 $x$ であるとき，$\mathcal{I}(x) = w^{-1}(x)$
- 正規表現の連結演算 $rs$ に対し，$\mathcal{I}(rs) = \mathcal{I}(r)\mathcal{I}(s)$
- 正規表現の和演算 $r|s$ に対し，$\mathcal{I}(r|s) = \mathcal{I}(r) \cup \mathcal{I}(s)$
- 反射的推移閉包 $r^*$ に対し，$\mathcal{I}(r^*) = \{(i,j) \mid \exists k_1, k_2, \ldots, k_{n-1},\ (i,k_1),(k_1,k_2),\ldots,(k_{n-1},j) \in \mathcal{I}(r)\}$.

最初の語に対する $w^{-1}$ の計算は，*Suffix Array* を用いて行うことになる．このアルゴリズムは，自然言語の用例の検索のように正規表現の基底となる構成語が一定の長さを持つようなケースには比較的有効である．また，計算の定義から明らかなように，連結を語と語の間にギャップを許すといったように自然に拡張することができる．

### 4.2 実装

上記アルゴリズムを実際に計算する方法について論じる．ここでの問題は，和と連結演算を効率的に行うことが可能な整数のペアの集合の実装方法が要となる．まず集合の保持の仕方であるが，要素が整数のペアであるので，直積順序 $\mathcal{O}_1$ を導入することで自然にソートすることが可能である．ただし，直積順序は第一要素と第二要素を反転させて得られる順序 $\mathcal{O}_2$ も考慮したい．そこで要素を直接ソートするのではなく，ソート結果を両順序に関し配列で保持することにする．以下にこの情報を利用した演算の効率的かつ自然な実装を示す．

まず和 $r|s$ であるが，$\mathcal{I}(r)$ と $\mathcal{I}(s)$ を $\mathcal{O}_1$ の順序で頭からマージして計算できる．これには両集合のサイズの和 $|\mathcal{I}(r)| + |\mathcal{I}(s)|$ の時間が必要となる．

[4] *Suffix Array* なら $\log n + |S(x)|$ で可能．

次に連結 $rs$ を考える．ここでは $\mathcal{I}(r)$ に関しては $\mathcal{O}_2$ を，$\mathcal{I}(s)$ に関しては $\mathcal{O}_1$ を用いる．要素のたどり方は和におけるマージと同様で，$\mathcal{I}(r)$ の要素を $\mathcal{O}_2$ に関して頭から，$\mathcal{I}(s)$ の要素を $\mathcal{O}_1$ に関して頭からなぞる．途中，$(i,k) \in \mathcal{I}(r)$，$(k,j) \in \mathcal{I}(s)$ なる要素が見つかるごとに $(i,j)$ を計算結果に加えていけばいい．計算にかかるコストはマージの分と結果の集合 $\mathcal{I}(rs)$ の大きさを足したもの，つまり $|\mathcal{I}(r)| + |\mathcal{I}(s)| + |\mathcal{I}(rs)|$ で抑えられる．

最後に反射的推移閉包 $r^*$ について考える．その定義から自然に分かるように，この計算は連結を複数回行うことによって実装できる．しかし，この連結の性質から，べき乗の各々の積（連結）において実際に新しい要素を生む可能性のある整数ペアは限られている．例えば $\mathcal{I}(r)$ の要素 $(i,j)$ で，第二成分 $j$ が他の要素の第一成分すべてと異なるならば，この要素を左から連結させても新しい要素は生まれない．つまり $\{(i,j)\}\mathcal{I}(r) = \emptyset$ が連結の定義から成り立つ．よって $\mathcal{I}(r)^2 = (\mathcal{I}(r) \setminus \{(i,j)\})^2$ であり，はじめからこのような要素を除いておけば計算の効率を上げることができる．また，この計算が $\mathcal{I}(r)$ を有効グラフの辺集合とみなしたときのグラフの各点からの到達可能性を調べることに相当することから分かるが，計算にかかるコストは上記連結の場合と同様である．

## 5 正規表現検索言語

上記アルゴリズムと実装方針に基づいて作成した正規表現検索のための言語を提示する．

```
<sentence> ::=
 let <variable> = # <word-list>;          |
 let <variable> = @ <v-list> [<gap>];     |
 let <variable> = + <variable> <variabl> |
 let <variable> = ^ <variable> <power>;  |
 print <variable>;
<variable> ::= 変数名
<v-list> ::= <variable> ... <variable>
<word-list> ::= "文字列" ... "文字列"
<power> ::= * | + | ’数字’
<gap> ::= ’数字’
```

この言語においては変数の型は唯一で，検索結果に対応する整数のペアの集合である．連結，和，べき乗（閉包も含めて）はそれぞれ `@`，`+`，`^` に対応する．使用方法の概略は以下の例のようになる．

```
let r = # "that" "are" "applied" "to";
let r = ^ r * 20;
print r;
```

ここで，第一行目の *let* 文を読むと，処理系は *Suffix Array* を利用し*”that” ... ”to”* の四つの文字列を含む部分を検索し，評価結果を変数 $r$ に格納する．つまり，$r = that|are|applied|to$ となる．第二行目で

は，そのように計算された部分式の反射的推移閉方が計算される．ただし最後の 20 は各連結で間に 20 字以下のギャップを許すという意味である．これにより $r^*$ が計算され変数 $r$ は更新される．結局全体として $(that|are|applied|to)^*$ がギャップ付きで計算されたことになる．このプログラムを，自然言語のタグ付きコーパスに適用したときに，最後の *print* 文が出力した結果を抜粋すると以下のようになる．

```
...
(4210,4237): are/VBP to/TO be/VB applied
(4210,4244): are/VBP to/TO be/VB applied/VBN to
(4218,4244): to/TO be/VB applied/VBN to
(4797,4822): are/VBP to/TO be/VB share
...
```

左はテキスト中の位置を示し，右はその部分の実際の文字列である．各単語間にギャップが許されているために，品詞タグが挿入されていても問題なく検索が行えていることが分かる．

## 6 結果と考察

前処理型の正規表現検索は，多くの場合に理論的観点で論じられてきた．その理由は，検索に利用するデータ構造が，テキストサイズと同等か，それ以上のメモリを必要とし，メモリには乗り切らないような実用的なサイズのデータに対して無力となってしまう点にある．本研究では *Suffix tree* よりはコンパクトな *Suffix Array* を使用してはいるが，この点に関して問題を抱えていることは同じである．これらのデータ構造の全体を主記憶上に置かず，必要な部分だけとってきて使用するなどの工夫も可能だが，余分な作業が増える分全体の検索速度が落ちるために前処理の価値が失われてしまう．また，線形にテキストを走査する *grep* のような手法が現在普通に利用されるような M 単位のテキスト，そして一般的なクエリーに対して十分なパフォーマンスを見せるという点も考慮せねばならない．CPU の性能向上とともに，テキストの前処理のような高速化なしでも検索が十分な速度で行える対象は拡大しているのである．

このような問題点がある一方で，今回の我々の手法には手法の簡潔さと柔軟性という面での利点がある．実装した正規表現検索用の言語を見ても分かるように，クエリーに対応するテキスト中の位置の集合という単一の型の上での計算に帰着していることから，連結においてギャップを許したり，さらには *Region Algebra* で行われているような高度な条件を自然な形で組み込むといった拡張が言語実装といったレベルで容易に行える．また，正規表現の構成に沿った形のアルゴリズムであるので，表現の部分評価ということが自然に行われている．これは，表現の評価結果，つまり検索の結果が上記のような型を与えられた対象として扱われているからである．よってこのような部分評価が意味をもつような応用を考えることが可能であり，さらに評価順の工夫による計算量の削減も考えられる．

## 7 まとめと今後の課題

正規表現検索を，対象テキストの前処理により高速化する方法についてまとめ，手法を提案した．特に，テキストの構造を自然に利用することで，正規表現検索を，インデックスのペアの集合という単一の型をもつ式の評価に帰着させた．結果としては計算の拡張可能性等に関する柔軟性を確認する一方，従来研究にも見られる主記憶サイズからの制約という問題を確認した．今後はインデックスを一文字単位ではなく，単語等の単位に変更したり，事前にフィルタリングをしてある程度候補を絞る，*Suffix Array* を分割することで主記憶に乗るように細かくするといった工夫をすることで実用的なデータに対して効率的に動くシステムを構築していくことが課題になる．また，本手法を含め，様々な手法がそれぞれどの分野の検索に有効なのかといった点をより明確にしていくことで，利用者が適切な手法を容易に選択できるようにしていかなければならない．

### 参考文献


[1] A. Blumer, J. Blumer and D. Haussler, The Smallest Automaton Recognizing The Subwords of A Text. *Theoretical Computer Science*, 40, 1985, pp. 31–35.

[2] C. L. A. Clarke, G. V. Cormack and F. J. Burkowski, An Algebra for Structured Text Search and A Framework for its implementation. *The computer Journal*, 38(1), 1995, pp. 43–56.

[3] R. Baeza-Yates and G. H. Gonnet, Fast Text Searching for Regular Expressions or Automaton Searching on Tries. *Journal of the ACM*, 43(6), 1996, pp. 915–936.

[4] G. Navarro and M. Raffinot, Flexible Pattern Matching in Strings. *Cambridge Univ. Press*, 2002.

[5] K. Thompson, Regular Expression Search Algorithm. *Communications of the ACM*, 11, 1968, pp. 419–422.

[6] P. Weiner, Linear Pattern Matching Algorithms, *IEEE 14th Annual Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[7] U. Manber and G. Myers, Suffix Arrays: A new method for on-line string searches, *SIAM J. Comput.*, 22(5), 1993, pp. 935–948, Oct.