LA-007

Arc-annotation付きテキストに対するパターン照合アルゴリズム
Pattern Matching Algorithm for Arc-annotated texts

喜田 拓也 *
Takuya Kida

1. はじめに

パターン照合技術は情報検索における重要な技術の一つである。近年、盛んに行われているテキスト・データマイニングの基礎技術としても用いられ、多くの研究者らによって効率のよいアルゴリズムの開発がなされている。

本分野におけるこれまでの研究では、いかに高速に照合を行うかが主な目的であった。この観点からこれまでは、データに関する知識を考慮せず、データを単なる文字列として扱うことで効率のよいパターン照合を実現してきた。しかし実用的に高度な検索を行うためには、データが明示的・暗示的に持っている構造を考慮した照合が要求される。たとえば、XMLファイルやHTMLファイルなどの半構造化データは、タグを単位とした木構造を内部表現に持つ。これに対し我々は、木構造を抽出することなしに高速に文字列照合を行う手法をこれまでに開発している[4]。また、様々な分野のテキスト情報に関する知識体系がシソーラスや分類階層といった形でデータベース化され、ネットワーク上から手に入るようになってきているという背景から、分類階層構造を考慮したパターン照合アルゴリズムの開発も行った[3]。

本稿では、Arc-annotation付きテキストに対するパターン照合アルゴリズムについて論じる。Arc-annotation付きテキストとは、テキスト中の二つの文字間に関係があることを示すアーク（arc）が付随したテキストである（図1）。このような構造を用いると、たとえば日本語の「かかりうけ」の構造などを明示したテキストデータが表現できる。より直接的な動機付けとしては、ゲノム情報処理における転移RNAの構造（図2）を考慮したパターン照合がある[5]。すなわちここでの問題は、テキストとパターンの列がそれぞれアークの情報を伴って与えられたとき、テキストの部分列がパターンとアークの形を保存しつつ一致するかどうかを答えることである。この問題は、一般的にはNP完全であることが示されている[1]。しかしながら、Grammら[2]は、どの二つのアークも同じ位置の文字を共有せず、また交差がない場合には、$O(nm)$のアルゴリズムが存在することを示した。ここで、n, mはそれぞれテキストの長さとパターンの長さである。ただし[2]のアルゴリズムには一部誤りがあるため、そのままでは動作しない。またこれまでの研究では、実験によるアルゴリズムの実際的な評価は行われていなかった。

本稿では、Grammらのアルゴリズムを元に、より簡潔で高速なアルゴリズムを提案する。また、これらのアルゴリズムを実際に実装し、速度比較した実験結果を示す。

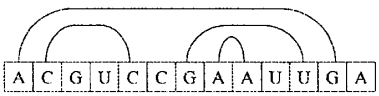

図 1: Arc-annotation付きテキスト

*北海道大学大学院情報科学研究科, Hokkaido University

2. 準備

Σを文字の有限集合とする。**列**（文字列）$S \in \Sigma^*$の長さを$|S|$で表し、長さが0の**空文字列**をεで表す。列$S \in \Sigma^*$のi文字目を$S[i]$で表し、i文字目からj文字目までの部分文字列を$S[i:j]$で表す。ただし、$j < i$の場合は$S[i:j] = \varepsilon$とする。列SとT（$|S| \leq |T|$）が与えられたとき、Tから$|T| - |S|$個の文字を間引くことでSが得られるならば、SをTの**部分列**という.

列Sに付随する**アーク**Aとは、範囲が$\{1, 2, \ldots, |S|\}$である整数の組の集合である。本稿では、各組$(i_L, i_R) \in A$について$i_L < i_R$が成り立っており、任意の組について同じ整数が含まれないと仮定する。列S上のアーク$(i_L, i_R) \in A$は、$S[i_L]$と$S[i_R]$を関係付けていることを示しており、文字$S[i_L]$と$S[i_R]$はそれぞれアークの**左端点**、**右端点**と呼ばれる。上の仮定から、どのアークも端点を共有しないことに注意する。また、集合Aの大きさも$|A|$と書く。

列Sがアークを持たないとき、そのような構造をplainと呼ぶ。アークが入れ子も交差もない場合にはchainと呼び、単に交差がない場合にはnestedと呼ぶ。アークが交差を含む場合にはcrossingと呼ばれる。すなわち、任意の二つのアーク$(i_L^1, i_R^1), (i_L^2, i_R^2) \in A$に対して、plainでは$i_R^1 < i_L^2$もしくは$i_R^2 < i_L^1$が成り立ち、nestedではさらに$i_L^2 < i_R^1 < i_R^2 \Leftrightarrow i_L^2 < i_L^1 < i_R^2$が成り立つ。

列S_1とS_2がそれぞれアークA_1とA_2を伴って与えられたとする。このとき、$S_1[i] = S_2[j]$（$1 \leq i \leq |S_1|$かつ$1 \leq j \leq |S_2|$）ならば、これを**ベースマッチ**（base match）と呼ぶ。また、$(i_L^1, i_R^1) \in A_1$と$(i_L^2, i_R^2) \in A_2$に対して、$S_1[i_L^1] = S_2[i_L^2]$かつ$S_1[i_R^1] = S_2[i_R^2]$が成り立つならば、これを**アークマッチ**（arc match）と呼ぶ。S_2がS_1の部分列である場合には、$\{1, 2, \ldots, |S_2|\}$から$\{1, 2, \ldots, |S_1|\}$の部分集合への一対一写像$M = \{(j, i_j) \mid 1 \leq j \leq |S_2|, 1 \leq i_j \leq |S_1|\}$が存在し、$(j, i_j) \in M$について$S_2[j] = S_1[i_j]$が成り立つ。いま、写像$M$によって$S_2$が$S_1$の部分列に対応付けされ、かつその$S_1$の部分列が$S_2$のアークを保存しているとき、$S_2$を$S_1$の

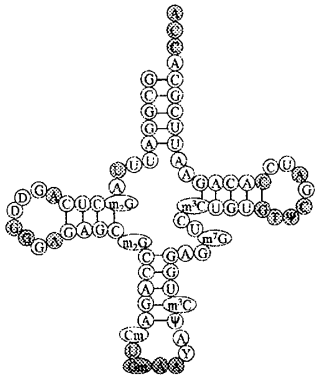
図 2: 酵母フェニルアラニンtRNAの構造

アーク保存部分列（arc-preserving subsequence）と呼ぶ。すなわち、任意の $(j_L, i_L), (j_R, i_R) \in M$ について、$(j_L, j_R) \in A_2 \Leftrightarrow (i_L, i_R) \in A_1$ が成り立つ。また、列 S_1 に対して、

$$I_1^{(k,\ell)} = \{i \mid k \le i \le \ell\} \setminus \bigcup_{\substack{(i_L,i_R)\in A_1 \\ \wedge k \le i_L < i_R \le \ell}} \{i' \mid i_L < i' < i_R\}$$

と定義する（$I_2^{(k,\ell)}$ も同様に定義する）。すなわち、$I_1^{(k,\ell)}$ は、部分文字列 $S_1[k,\ell]$ の内部でアークに囲まれた部分を除いた文字の位置の集合である。

関数 $\mathrm{maxaps}(S_1[i_1 : i_2], S_2[j_1 : j_2])$ は、$S_2[j_1 : j']$ が $S_1[i_1 : i_2]$ のアーク保存部分列となるような最大の j'（$j_1 \le j' \le j_2$）を返すものと定義する。もし、そのような j' が存在しない場合には $j_1 - 1$ を返す。

アーク保存部分列問題　アーク保存部分列問題（Arc-preserving subsequence problem）とは、長さ n の列 S_1 と長さ m の列 S_2 がそれぞれアーク A_1 と A_2 を伴って与えられたとき、S_2 が S_1 のアーク保存部分列であるか否かを答える問題である。便宜的に、列 S_1 をテキスト、列 S_2 をパターンと呼ぶ。テキストとパターンそれぞれのアークの構造によって問題の質が異なるため、その組み合わせを APS(TYPE1, TYPE2) で表し、前者 TYPE1 をテキストの構造、後者 TYPE2 をパターンの構造とする。たとえば、APS(nested, chain) はテキストの構造が nested で、パターンの構造が chain であることを示す。当然ながら、制限の少ない構造は、より制限の多い構造の問題を含んでいる。

3. GGN アルゴリズム

Gramm ら [2] は、APS(nested, nested) を $O(nm)$ で解くアルゴリズムを提案した。彼らのアイデアでは、まずテキストとパターンのアークで囲まれた部分文字列どうしの maxaps を動的計画法を用いて計算し、最後に S_1、S_2 全体どうしの maxaps を計算する。手順としては APS(nested, plain) 用のアルゴリズムを土台に APS(nested, chain) へと拡張し、それらの結果を踏まえて APS(nested, nested) を解くアルゴリズムを得ている。Gramm らのアルゴリズム（以下 GGN アルゴリズム）を大まかに解析すると、任意の i, j $(1 \le i \le n, 1 \le j \le m)$ について $S_1[i]$ と $S_2[j]$ を比較することになるので、計算時間は $O(nm)$ である。また、動的計画法で必要となるテーブルのサイズは $O(|A_1|m)$ である。GGN アルゴリズムに修正を加えたものを図 3 に示す。

4. 改良型 GGN アルゴリズム

GGN アルゴリズムは、ボトムアップ型の動的計画法である。すべての i, j $(1 \le i \le n, 1 \le j \le m)$ の組み合わせについて $S_1[i]$ と $S_2[j]$ を比較するが、最終的に $\mathrm{maxaps}(S_1[1 : n], S_2[1 : m]) = m$ かどうかだけを知りたいのであれば、大半の計算が無駄であることが観察できる。このアルゴリズムをトップダウン型の動的計画法に変形し、真に必要な値のみを保存するようにすれば、計算を大幅に省くことができる。また、ベースマッチしなければアークマッチもしないという性質を用いることでも計算の枝狩りができる。さらに、GGN アルゴリ

```
function maxaps_np(S1[i1 : i2], S2[j1 : j2]) {
  if (S1[i1 : i2] = ε もしくは S2[j1 : j2] = ε) return j1 − 1;
  else if (i1 = i2) {
    if (S1[i1] = S2[j1]) return j1;
    else return j1 − 1;
  } else if (j1 = j2 かつ i1 < i2) {
    if (S1[i1] = S2[j1]) return j1;
    else return maxaps_np(S1[i1 + 1 : i2], S2[j1 : j1]);
  } else if (i1 < i2 かつ j1 < j2) {
    if (S1[i1] がアーク (iL, iR) ∈ A1 の左端点) {
      return maxaps_np(S1[iR + 1 : i2], S2[T(i1, j1) + 1 : j2]);
    } else if (S1[i1] = S2[j1]) {
      return maxaps_np(S1[i1 + 1 : i2], S2[j1 + 1 : j2]);
    } else {
      return maxaps_np(S1[i1 + 1 : i2], S2[j1 : j2]);
    }
  }
}
function maxaps_nc(S1[i1 : i2], S2[j1 : j2]) {
  if (S1[i1 : i2] = ε もしくは S2[j1 : j2] = ε) return j1 − 1;
  else if (i1 = i2) {
    if (S1[i1] = S2[j1] かつ S2[j1] が端点ではない) return j1;
    else return j1 − 1;
  } else if (i1 < i2 かつ j1 = j2) {
    if (S2[j1] が端点ではない) return j1 − 1;
    else if (S1[i1] = S2[j1]) return j1;
    else return maxaps_nc(S1[i1 + 1 : i2], S2[j1 : j1]);
  } else if (i1 < i2 かつ j1 < j2 かつ
          S1[i1] も S2[j1] も端点ではない) {
    if (S1[i1] = S2[j1]) return maxaps_nc(S1[i1 + 1 : i2], S2[j1 + 1 : j2]);
    else return maxaps_nc(S1[i1 + 1 : i2], S2[j1 : j2]);
  } else if (i1 < i2 かつ j1 < j2 かつ
          S2[j1] が端点かつ S1[i1] は端点でない) {
    return maxaps_nc(S1[i1 + 1 : i2], S2[j1 : j2]);
  } else if (i1 < i2 かつ j1 < j2 かつ
          S1[i1] がアーク (iL, iR) ∈ A1 の左端点) {
    if (T(i1, j1) が未定義) {
      T(i1, j1) = max(maxaps_nc(S1[i1 : iR − 1], S2[j1 : j2]),
          maxaps_nc(S1[i1 + 1 : iR], S2[j1 : j2]));
    }
    return maxaps_nc(S1[iR + 1 : i2], S2[T(i1, j1) + 1 : j2]);
  }
}
procedure GGNalgorithm(S1, A1, S2, A2) {
  for each (jL, jR) ∈ A2 (左端点の降順に) {
    for each j ∈ I2^(jL+1, jR−1) {
      for each (iL, iR) ∈ A1 (左端点の降順に) {
        if ((jL, jR) ∈ A2 は最も内側のアーク) {
          T(iL, j) = max(maxaps_np(S1[iL : iR − 1], S2[j : jR − 1]),
              maxaps_np(S1[iL + 1 : iR], S2[j : jR − 1]));
        } else {
          T(iL, j) = max(maxaps_nc(S1[iL : iR − 1], S2[j : jR − 1]),
              maxaps_nc(S1[iL + 1 : iR], S2[j : jR − 1]));
        }
      }
    }
    for each (iL, iR) ∈ A1 (左端点の降順に) {
      if ((iL, iR) ∈ A1 は maxaps_nc(S1[iL : iR], S2[jL : jR]) = jR
            となる最も内側のアーク) {
        T(iL, jL) = jR;
      } else {
        T(iL, jL) = maxaps_nc(S1[iL + 1 : iR], S2[jL : m]);
      }
    }
  }
  for each j ∈ I2^(1,m) (降順に) {
    for each (iL, iR) ∈ A1 (左端点の降順に) {
      T(iL, j) = max(maxaps_nc(S1[iL + 1 : iR], S2[j : m]),
          maxaps_nc(S1[iL : iR − 1], S2[j : m]));
    }
  }
  if (maxaps_nc(S1[1 : n], S2[1 : m]) = m) {
    print 'S2 is an aps of S1';
  } else {
    print 'S2 is an aps of S1';
  }
}
```

図 3: APS(nested, nested) を解く GGN アルゴリズム

ズムでは APS(nested, plain) の場合での maxaps である $\text{maxaps}_{\text{np}}$ と、APS(nested, chain) の場合の $\text{maxaps}_{\text{nc}}$ をそれぞれ用意して用いているが、本質的には $\text{maxaps}_{\text{nc}}$ は $\text{maxaps}_{\text{np}}$ の処理を含むはずである。これらのアイデアを元に改良を加えると、図 4 のアルゴリズム (Faster GGN アルゴリズム) が得られる。

5. 実験結果

上述したアルゴリズムを C 言語で実装し、$\Sigma = \{A, C, G, U\}$ 上でランダムに生成したデータを用いていくつかの速度比較実験を行った。使用したマシンは DELL Precision650(Intel Xeon3.06GHz Dual プロセッサ、メモリ 3.5GB) で、Windows XP 上の Cygwin 環境で実行した。

図 5 は、$m = |S_2| = 20$、$|A_2| = 4$ のパターンに対して、テキスト側の $n = |S_1|$ を 100 ～ 1000 まで変化させた場合のグラフである。$|A_1|$ は n の 20% としている。図 6 は、$n = |S_1| = 1000$、$|A_1| = 100$ のテキストに対して、パターン側の $m = |S_2|$ を 10 ～ 100 まで変化させた場合のグラフである。$|A_2|$ は m の 20% としている。図 7 は、$n = |S_1| = 1000$、$|A_1| = 100$ のテキストに対して、パターン側の $|A_2|$ を 0 ～ 20 まで変化させた場合のグラフである。$m = |S_2|$ は 50 で固定している。図 8 は、$n = |S_1| = 1000$、$|A_1| = 100$ のテキストに対して、パターン側の入れ子の深さの最大値を 1 ～ 10 まで変化させた場合のグラフである。$|A_2| = 10$、$m = |S_2| = 50$ としている。すべての実験において、同条件のテキスト列を 1000 個用意し、各々5 つの異なるパターンに対してアルゴリズムを実行した。それらの CPU 時間を計測し、パターン一つあたりの実行時間の平均を求めた。

6. おわりに

実験結果より、提案アルゴリズムによって、すべての場合において実行速度が 2 ～ 5 倍以上改善されることがわかった。また、パターンの長さやアークの数、入れ子の深さに影響されにくくなった。

APS(crossing, plain) については NP 完全かどうかが判っておらず、入力の多項式時間アルゴリズムが期待される。また、現実のデータにおいては n, m は小さい場合が多いため、APS(crossing, crossing) でも実用的な時間で解ける可能性がある。そのようなアルゴリズムの開発は、実際の問題を解決するうえで有益である。

参考文献

[1] P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proc. 10th CPM*, Vol. 1645 of *LNCS*, pp. 270–280. Springer, 1999.

[2] J. Gramm, J. Guo, and R. Niedermeier. Pattern matching for arc-annotated sequences. In *Proc. 22nd FSTTCS*, Vol. 2556 of *LNCS*, pp. 182–193. Springer, 2002.

[3] T. Kida and H. Arimura. Pattern matching with taxonomic information. In *Proc. Asia Information Retrieval Symposium*, pp. 265–268, October 2004.


```
function maxaps(S1[i1 : i2], S2[j1 : j2]) {
  if (S1[i1 : i2] = ε もしくは S2[j1 : j2] = ε) {
    return j1 − 1;
  }
  for (i = i1, j = j1; i ≤ i2 かつ j ≤ j2; i++) {
    if (S1[i] ≠ S2[j]) {
      continue;
    }
    if (S2[j] が端点ではない) {
      if (S1[i] がアーク (iL, iR) ∈ A1 の左端点ではない) {
        j++;
      } else {
        if (T(i, j) が未定義) {
          T(i, j) = max(maxaps(S1[i : iR − 1], S2[j : j2]),
                maxaps(S1[i + 1 : iR], S2[j : j2]));
        }
        j = T(i, j) + 1;
        i = iR;
      }
    } else {
      if (S2[j] がアーク (jL, jR) ∈ A2 の左端点) {
        if (S1[i] がアーク (iL, iR) ∈ A1 の左端点
              かつ S1[iR] = S2[jR]) {
          t1 = maxaps(S1[i + 1 : iR − 1], S2[j + 1 : jR − 1]);
          t2 = maxaps(S1[i + 1 : iR], S2[j : j2]);
          if (t1 = jR − 1 かつ t1 > t2) {
            T(i, j) = t1 + 1;
          } else {
            T(i, j) = t2;
          }
          j = T(i, j) + 1;
          i = iR;
        }
      }
    }
  }
  return j − 1;
}

procedure FGGNalgorithm(S1, A1, S2, A2) {
  if (maxaps(S1[1 : n], S2[1 : m]) = m) {
    print 'S2 is an aps of S1';
  } else {
    print 'S2 is an aps of S1';
  }
}
```

図 4: Faster GGN(FGGN) アルゴリズム

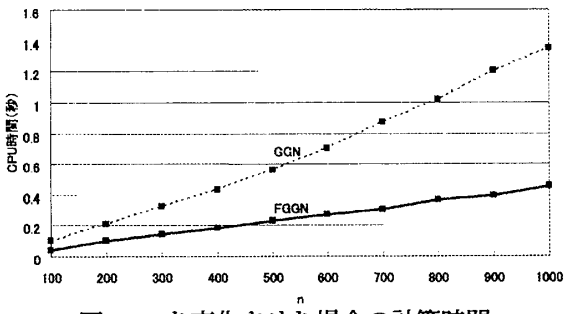

図 5: n を変化させた場合の計算時間

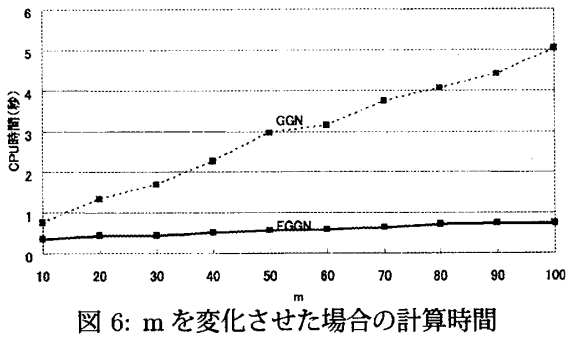

図 6: m を変化させた場合の計算時間

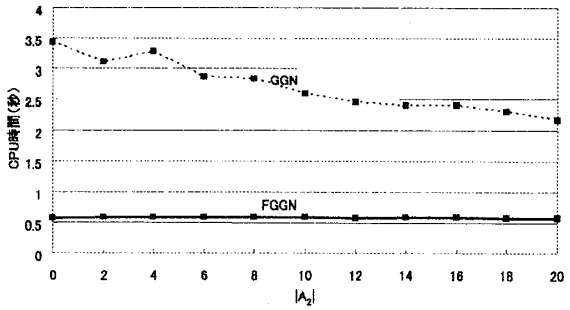

図 7: $|A_2|$ を変化させた場合の計算時間

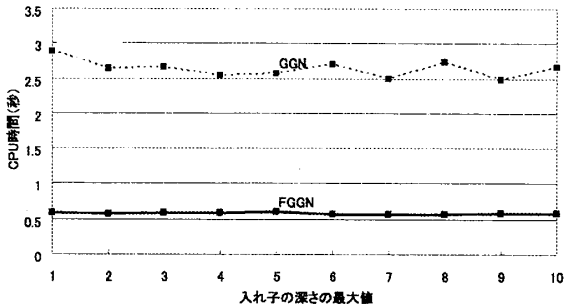

図 8: 入れ子の深さを変化させた場合の計算時間

[4] M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *Proc. 9th International Symposium on String Processing and Information Retrieval*, Vol. 2476 of *LNCS*, pp. 170–186. Springer, September 2002.

[5] S. Vialette. On the computational complexity of 2-interval pattern matching problems. *Theoretical Computer Science*, 312(2-3):223–249, January 2004.