DEIM2020 D1-1(day1 p22)

全部分文字列に対する出現間隔の分析のためのアルゴリズム実装

小原　佑斗† 　三浦　準也† 　梅村　恭司†

† 豊橋技術科学大学 情報・知能工学系　〒 441–8580 愛知県豊橋市天伯町雲雀ケ丘 1–1
E-mail: †{y173321,j183370}@edu.tut.ac.jp, ††umemura@tut.jp

あらまし　ある単語の文書中での出現間隔の情報は重要度に関係するものであるが，本研究では，単語だけでなく，文書中のあらゆる部分文字列について，その出現間隔に基づいた複数の統計量を $O(n \log n)$ 時間（n：文書長）で計算するアルゴリズムを提案する．このアルゴリズムは文書中に出現する重要な文字列を特定するタスクに応用できる．現在，時間計算量 $O(n(\log n)^2)$ の実装ができている．これを，時間計算量 $O(n^2)$ の単純なアルゴリズムと比較することで，実行速度の検証を行う．

キーワード　接尾辞木，キーワード抽出，string statistics problem

1 はじめに

意味のある単語は，一度現れたら，あらわれやすくなる [1] という報告があり，日本語においてもキーワードの抽出をこの考え方で行えるという報告 [2] がある．これらの方法は，回数を指定して，その回数以上，対象の単語が現れる文書数をもとに，繰り返しの出現の傾向を分析している。この文書数を求めるアルゴリズムとして，接尾辞配列を用いた方法 [3] があり，これはすべての部分文字列について一斉に求めることを特徴としている方法である．繰り返しの傾向の分析として，われわれは，文書の区切りをあたえるかわりに，指定した範囲のなかでの繰り返しの回数を計測することに興味をもった．ここで，キーワードの抽出を行うには，繰り返しの傾向の高い文字列を探すタスクになり，対象となる文書のあらゆる文字列を対象にして統計量を計算し，繰り返しの傾向の高い文字列を選び出すことが必要になる．分析の対象となる文字列の数は多く，それごとに対象文書を調べていく方法では計算時間が問題になる．

このような背景から，前処理を許すことによって，全部分文字列について，あたえられた数より近い再出現の個数を求める手法を模索し始めた．調査をしていく過程で，重なりがない状態での出現数を，一斉にもとめる方法があることがわかった [4]．重なりの検出は，近くに現れるケースを検出する機能を含んでいると考えられるため，その実装を調査した．この方法の再実装を試みた．再実装の途中で，むしろわれわれの問題のほうが単純にもとめることができることが判明したので，本論文ではその方法を報告する．

2 出現間隔に基づいた統計量

本論文では，文字列 S 中の文字列 x の出現のうち，その直前の出現から k 文字以内のものの個数 $c_{S,k}(x)$ について取り扱う．これを求める単純なアルゴリズムを C 言語で記述すると図 1 のようになる．

例として，$S =$ “aabaaabaab”, $k = 3, x =$ “aab” の場合を考える．x は S 中の $\{0, 4, 7\}$ 文字目に出現する（先頭を 0 文字目としている）．この出現位置のリストを，図 2 のように，となりあう出現を実線や点線でつないだ図で表現する．実線は前後の出現位置の差が k 以下であることを意味し，点線はそうでないことを意味する．この実線の本数こそが $c_{S,k}(x)$ であるため，この場合は $c_{S,k}(x) = 1$ である．

```
int count(char *S, int k, char *x) {
  char *p = strstr(S, x), *q;
  if (p == NULL) return 0;
  int c = 0;
  while ((q = strstr(p + 1, x)) != NULL) {
    if (q - p <= k) c++;
    p = q;
  }
  return c;
}
```

図 1　単純なアルゴリズムで $c_{S,k}(x)$ を求める C プログラム

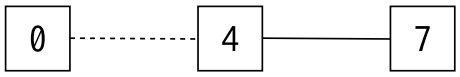

図 2　$S =$ “aabaaabaab”, $k = 3$ の場合の，$x =$ “aab” の出現位置リスト

3 準　　備

文字列 S の i $(0 \leq i < |S|)$ 文字目以降を切り出した文字列を，S の i 番目の接尾辞とよぶ．文字列 S の接尾辞木は，S\$（\$ は S 中に登場しない文字）のすべての接尾辞を木構造で表現したデータ構造である．ここでは例として，$S =$ “aabaabaabb” の接尾辞木を考える．これを図 3 に示す．各エッジには S の部分文字列がラベルとして割り当てられている．各葉ノードに割り当てられている数値 i は，根からその葉ノードまでのラベルをすべて連結してできる文字列が，S の i 番目の接尾辞となっていることを表す．また，ノードの個数は $O(|S|)$ である．

S の接尾辞木を構築すると，文字列 x が S 中の何文字目に出現するかを調べることができる．たとえば，$x =$ “aab” のと

きは，根ノードから “a”→“a”→“b” というラベルのエッジを順にたどると中間ノード v_2 にたどり着く．この v_2 以下に存在する葉ノード $\{0,4,7\}$ が x の S 中での出現位置である．以降，あるノード v 以下に存在する葉ノードの集合は v の葉リストとよび，$\mathrm{LL}(v)$ と表記する．$x =$ “aba” の出現位置を調べるときは，根ノードから “a”→“b”→“a” とたどりたいところだが，最後の “a” というラベルのエッジが存在しない．この場合は “a” という接頭辞をもつ “aa” ラベルのエッジをたどる．このようにして v_4 にたどり着くと，出現位置は $\mathrm{LL}(v_4) = \{1,5\}$ であることがわかる．これは，“aba” と “abaa” が S 中で全く同じ位置に出現するということも示している．

先述したように，$c_{S,k}(x)$ の値は x の出現位置がわかれば一意に求まるため，“aba” と “abaa” のような，根ノードからたどって同じノードにたどり着く文字列は $c_{S,k}(x)$ の値も等しい．以降，根ノードからたどってノード v にたどり着く文字列 x の $c_{S,k}(x)$ は，$c_{S,k}(v)$ とも表記する．すべてのノード v について $c_{S,k}(v)$ を求めれば，S のあらゆる部分文字列 x についての $c_{S,k}(x)$ を求めたことになる．これを図 4 のようにノードに格納しておけば，$c_{S,k}(x)$ の値は $O(|x|)$ 時間で取り出すことができる．

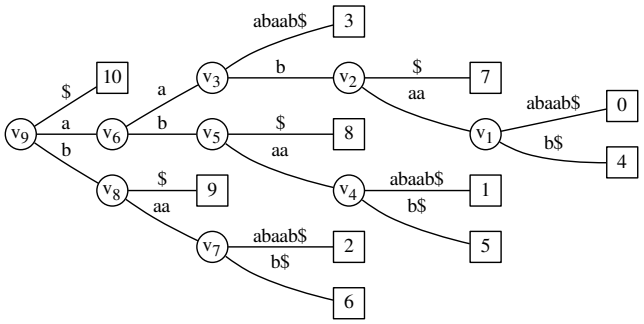

図 3　$S =$ “aabaaabaab” の接尾辞木

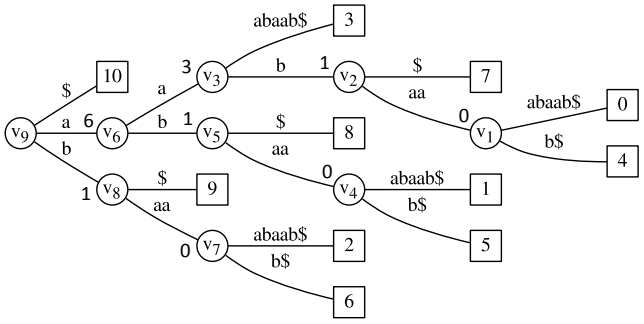

図 4　$S =$ “aabaaabaab” の接尾辞木の，各ノード v についての $c_{S,3}(v)$ の値

4　アルゴリズム

4.1　基本的なアイデア

S の接尾辞木を構築した後，接尾辞木の各ノード v について $c_{S,k}(v)$ を求めるアルゴリズムを考える．単純には，すべての v について $\mathrm{LL}(v)$ を求め，これを昇順にソートし，1 つ前の要素との差分が k 以下となっている要素を数える方法が考えられる．しかし，この方法は最悪の場合，時間計算量が $O(|S|^2)$ 以上になってしまう．具体的には，$S =$ “aaaaa” のように，S の文字がすべて同じ文字で構成される場合である．この場合，S の接尾辞木は図 5 のようになる．単純な方法では，各ノード v_i ごとに $|\mathrm{LL}(v_i)| = i+1$ 個の葉ノードを参照する必要があるため，のべ $O(|S|^2)$ 個の葉ノードを参照することになる．よって，時間計算量は少なくとも $O(|S|^2)$ となる．

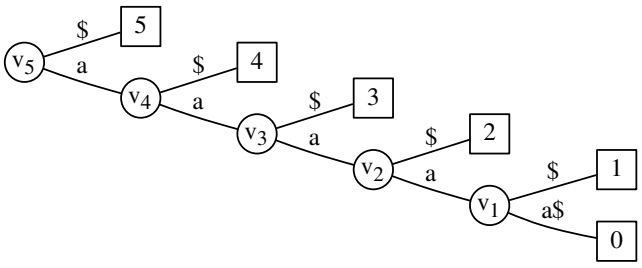

図 5　$S =$ “aaaaa” の接尾辞木

これを回避するためには，$\mathrm{LL}(v) \supset \mathrm{LL}((v \text{ の子ノード}))$ であることを利用し，子ノードで求めた葉リストの情報を再利用する必要がある．このように，接尾辞木をボトムアップ的に走査し，ノード v の葉リスト $\mathrm{LL}(v)$ を，v の子ノードの葉リストを利用して求め，その過程で所望の値を求めるというものが，ベースとなるアルゴリズム [4] の基本的なアイデアである．

4.2　$\mathrm{LL}(v)$ の求め方

[4] のアルゴリズムの一部を利用すれば，すべての v について $\mathrm{LL}(v)$ を求める操作を $O(|S|(\log|S|)^2)$ 時間で終えることができる．この際，$\mathrm{LL}(v)$ を表現するために，赤黒木などの，ソート済みリストを表現するデータ構造が必要となる．このデータ構造は，要素数を n として，新たな要素を挿入する操作，ある要素の直前・直後の要素を求める操作を $O(\log n)$ 時間で行えることが保証されているものとする．

具体的には，各ノード v について，以下の手順で $\mathrm{LL}(v)$ を求める．

（1） v の子ノード v_c のうち，もっとも $|\mathrm{LL}(v)|$ が大きいものを探し，v_{cmax} とする．

（2） v 以下の葉ノードのうち，$\mathrm{LL}(v_{\mathrm{cmax}})$ に含まれないものをすべて $\mathrm{LL}(v_{\mathrm{cmax}})$ に挿入する．

[4] では $\mathrm{LL}(v)$ を表現するために level-linked (2,4)-trees というデータ構造を用い，葉ノードの挿入方法も工夫することで，この操作の時間計算量を $O(|S|\log|S|)$ にできることが述べられている．しかし，実装コストが高いこと，計算量に大きな差がないことから，現状は $O(|S|(\log|S|)^2)$ のもののみ実装している．

4.3　$c_{S,k}(v)$ の求め方

例として，また図 3 の接尾辞木について考える．前述の手順に従えば，$\mathrm{LL}(v_3)$ を求める際は，$\mathrm{LL}(v_2)$ に，v_3 直下の葉ノードである 3 を挿入する操作が必要となる．このときのようすを，

図 2 のようなブロック図を用いて表したものが図 6, 7, 8 である．これらは k が異なるそれぞれの場合について，3 を挿入した後に葉リストの実線の本数が変化するようすを示している．注意すべきは，実線の本数が変化するのは挿入した要素の左右のみということである．図 6 は $k = 2$ の場合で，3 を挿入すると左の 0 との間は実線にならないが，右の 4 との間は差が k 以下となるため，実線になる．図 7 は $k = 3$ の場合で，3 を挿入すると左右の両方とも差が k 以下となるので，両側が実線となる．図 8 は $k = 10$ の場合である．前述した 2 つのパターンと違うのは，挿入先の 1–10 間がもともと実線で結ばれている点である．この場合は 1–10 間の実線を 1 度点線にし，その上で 3 を挿入し，その左右を実線にしたと考えることができる．

以上より，ノード v の $c_{S,k}(v)$ を求める際は，$c_{S,k}(v_{c\max})$ を初期値とし，葉リストに対して要素 l を 1 つ挿入するごとに，l とその 1 つ左の要素との差が k 以下であれば+1，l とその 1 つ右の要素との差が k 以下であれば+1，l の左と右の要素の差が k 以下であれば–1，とカウントしていけばよい．

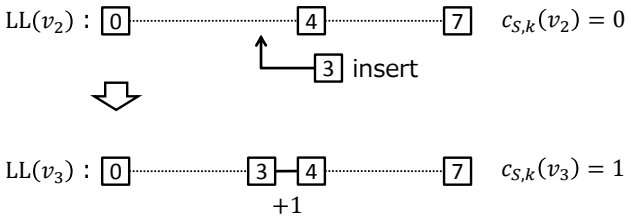

図 6　$k = 2$ の場合の，LL(v_3) を求める操作

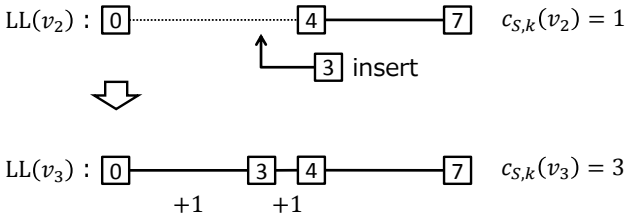

図 7　$k = 3$ の場合の，LL(v_3) を求める操作

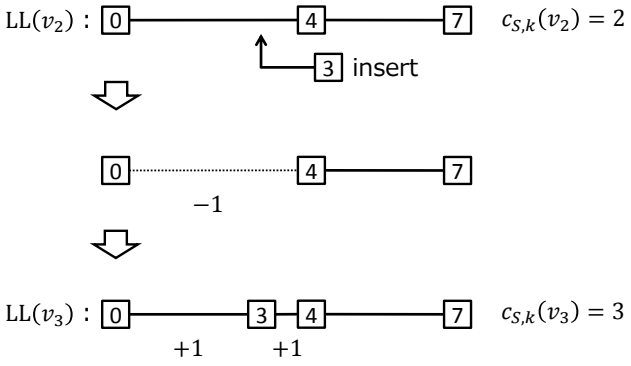

図 8　$k = 10$ の場合の，LL(v_3) を求める操作

5　計算量の解析

接尾辞木は多分木であるため，その中間ノードは 3 つ以上の子ノードをもつ場合がある．そのようなノードでは，図 9 のように，空文字列 ϵ のラベルをもつエッジが存在すると考えることで，接尾辞木を二分木として表現することができる．[4] のアルゴリズムでは，このように接尾辞木を二分木として考えた場合の，1 つの中間ノードあたりの計算量が $O(|\mathrm{small}(v)| \cdot \log(|v|/|\mathrm{small}(v)|))$ ($|v|$ は v 以下の葉ノードの数 $|\mathrm{LL}(v)|$，$\mathrm{small}(v)$ は v の 2 つの子ノード v_c のうち，$|v_c|$ が小さいもの）であるため，以下の定理 1 を証明することで，全体の時間計算量が $O(n \log n)$ となることが示されている．

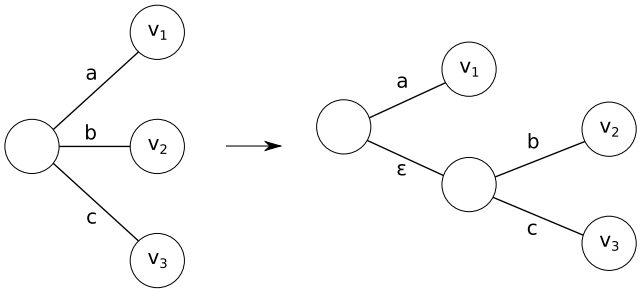

図 9　接尾辞木の二分木への変換（$|v_1| > |v_2| > |v_3|$）

［定理 1］　（[4] より引用）n 個の葉をもつ二分木を T とする．T の中間ノード v について $c_v = |\mathrm{small}(v)| \cdot \log(|v|/|\mathrm{small}(v)|)$，葉ノード v について $c_v = 0$ とすると，

$$\sum_{v \in T} c_v \leqq n \log n \tag{1}$$

同様に，接尾辞木を二分木として考えた場合，本論文のアルゴリズムでは，各中間ノード v で葉リストに対して $|\mathrm{small}(v)|$ 回の挿入処理を行う．1 回の挿入に必要な計算量は最大でも $O(\log(|v|))$ 時間であるため，1 つの中間ノードあたりの計算量は大きく見積もっても $O(|\mathrm{small}(v)| \cdot \log(|v|))$ 時間である．よって，以下の定理 2 が正しければ，全体の計算量は $O(n(\log n)^2)$ である．

［定理 2］　n 個の葉をもつ二分木を T とする．T の中間ノード v について $c_v = |\mathrm{small}(v)| \cdot \log(|v|)$，葉ノード v について $c_v = 0$ とすると，

$$\sum_{v \in T} c_v \leqq n(\log n)^2 \tag{2}$$

証明は定理 1 と同様に，数学的帰納法を用いて行う．まず，$|T| = 1$ の場合，(2) は明らかに成り立つことがわかる．ここで，$|T| = 2$ から $|T| = n - 1$ までのすべての場合で (2) が成り立つと仮定したときに，$|T| = n$ で (2) が成り立つかどうかを考える．根の子ノード以下の葉の数をそれぞれ $k, n-k$ $(1 \leq k \leq \frac{n}{2})$ とすると，根ノードを除いた c_v の合計は $k(\log k)^2 + (n-k)(\log(n-k))^2$ 以下となるため，

$$k(\log k)^2 + (n-k)(\log(n-k))^2 + k \log n \leqq n(\log n)^2 \tag{3}$$

であれば (2) が成り立つといえる．ここで，(3) の左辺を k で二階微分した二次導関数は $\frac{2}{k}(\log k+1)+\frac{2}{n-k}(\log(n-k)+1))$ であり，$1 \leqq k \leqq \frac{n}{2}$ では常に正の値となる．よって，一次導関数の $k=1$ での値が正であっても負であっても $1 \leqq k \leqq \frac{n}{2}$ で極大値をもつことはないため，(3) の左辺は境界である $k=1$ もしくは $k=\frac{n}{2}$ で最大値をとる．$k=1$ のとき，(3) は $(n-1)(\log(n-1))^2+\log n \leqq n(\log n)^2$ となり成り立つ．$k=\frac{n}{2}$ のとき，(3) は $n(\log n)^2+n(\log 2)^2+\frac{n}{2}\log n-2n(\log 2)(\log n) \leqq n(\log n)^2$ となり成り立つ．以上より，任意の n について (2) が成り立つ．

6 速度検証

本論文で提案したアルゴリズムを実装し，これが想定通り高速に動作するかを検証する．なお，実装上では，接尾辞木をよりコンパクトに表現できる，接尾辞配列というデータ構造を用いた．接尾辞木を用いるアルゴリズムは接尾辞配列を用いてシミュレート可能であることが示されている [5]．

比較対象は，4.1 節で説明した，最悪計算量 $O(n^2)$ の手法（単純法）とする．使用するデータは，

（1） 欽定訳聖書（4,332,557 字）

（2） 単純法の最悪ケース（“aaaaa..”）（10000〜50000 字）

とする．これらのデータに対してそれぞれのアルゴリズムを適用し，全部分文字列 x について $c_{S,k}(x)$ を求める．k の値は実行時間にほぼ影響しないため，$k=100$ で固定とする．

6.1 結果：欽定訳聖書

単純法の実行時間が 12.83 秒だったのに対し，提案手法の実行時間は 21.31 秒だった．このことから，自然言語に対しては，単純法でも十分高速に動作することが示唆された．この実験では，提案手法はアルゴリズムの複雑さにより，定数倍レベルで遅くなってしまったと考えられる．

6.2 結果：単純法の最悪ケース

結果を表 1 に示す．単純法は自然言語の場合と比べて明らかに遅くなり，データ長に対して 2 乗オーダーで実行時間が増加している．これに対し，提案手法ではそのようなことはなく，極端なケースでも十分高速に動作することが確認できた．

表 1　単純法の最悪ケースでの実行時間（単位：秒）

データ長	単純法	提案手法
10000	4.89	0.06
20000	20.99	0.09
30000	48.26	0.08
40000	89.51	0.14
50000	141.2	0.16

7 string statistics のアルゴリズム [4] との大きな相違点

string statistics においては，接尾辞木を拡張して，重複を許さない出現数の切れ目に相当するノードを追加して処理をすることが必要であったが，このアルゴリズムはそれが必要ではない．一方で，マージをするときに，出現数の少ないノードからもとまる出現場所を，出現数の多いノードからもとまる出現場所集合を表現するデータに挿入していくという操作は共通のものである．この方法が，計算のオーダを小さくすることは，直感的には明らかでないが，string statistics のアルゴリズムにおける計算量のオーダの計算をなぞることで，計算量が小さくなることがわかる．一方で，典型的な自然言語を対象とした実際の計算時間でも高速にするには，データの表現とアルゴリズムをさらに改良しなければいけないことはわかっている．

8 まとめ

接尾辞木の各ノードにおける葉リストのマージの方法を注意深く設計することによって，接尾辞木をボトムアップ的に走査する手順で，与えられた間隔以下で再出現するケースの発生数を求められることを報告した．その方法のコードを実装して，動作を確認した．

文献

[1] Kenneth W. Church. Empirical Estimates of Adaptation: The Chance of Two Noriegas is Closer to $p/2$ than p^2. In *Proceedings of the 18th Conference on Computational Linguistics - Volume 1*, p. 180186, 2000.

[2] 武田善行, 梅村恭司. キーワード抽出を実現する文書頻度分析. 計量国語学会, Vol. 23, No. 2, pp. 65–90, 2001.

[3] 梅村恭司, 真田亜希子. 文字列を κ 回以上含む文書数の計数アルゴリズム. 自然言語処理, Vol. 9, No. 5, pp. 43–70, 2002.

[4] Gerth Stølting Brodal, Rune B Lyngsø, Anna Östlin, and Christian N S Pedersen. Solving the String Statistics Problem in Time $\mathcal{O}(n \log n)$. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pp. 728–739, 2002.

[5] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, Vol. 2, No. 1, pp. 53–86, 2004.