DEIM Forum 2009 E8-6

文字列ストリームのための近似的な索引構造

上村　卓史†　　喜田　拓也†　　有村　博紀†

† 北海道大学大学院情報科学研究科　〒 060–0814 札幌市北区北 14 条西 9 丁目
E-mail: †{tue,kida,arim}@ist.hokudai.ac.jp

あらまし　与えられたテキスト中における部分文字列の索引化は，多くの応用を持つ重要な問題である．しかし，Web のような莫大かつ増え続ける文字列ストリームデータに対しオンラインで完全な索引を構築することは，計算量，領域の観点からきわめて困難である．本稿では，現実の主記憶容量にあわせて索引サイズを設定でき，任意の文字列に対してその出現頻度の近似値を計算可能なデータ構造を提案する．
キーワード　文字列出現頻度推定，接尾辞木

An Approximate Substring Index for Text Stream

Takashi UEMURA†, Takuya KIDA†, and Hiroki ARIMURA†

† Graduate School of Information Science and Technology, Hokkaido University
N14 W9, Sapporo 060–0814, Japan
E-mail: †{tue,kida,arim}@ist.hokudai.ac.jp

Abstract　In text processing applications, we often need to know the frequency of a query in the given massive text. However, it is difficult to construct an index structure in the main memory which computes the true frequency of any query in the text. In this paper, we present a compact index structure which uses a constant size of main memory and estimates the frequency of any query in the text.
Key words　substring selectivity estimation, suffix trees

1. はじめに

インターネットをはじめとした様々な応用において，大規模な文字列データを扱う機会が増えている．中でも，与えられた大きな文字列 (テキスト)T の中に，文字列 (クエリ)P が何回出現するかという問題は，きわめて基本的かつ実際の応用においても重要である．特にクエリの個数が多い場合，高速な計算には索引構造の構築が不可欠である．また，データ圧縮や機械学習をはじめとして，特定の言語を前提とせず任意の部分文字列を考慮する応用も多い．さらに，データによっては，オフラインでのバッチ処理的な手法ではなく，オンラインでの逐次処理が望ましい．本研究では，データ圧縮のほか，ウェブ上の文字列データに対するストリーム処理に向けた部分文字列索引の構築について考察する．

接尾辞木 [4] は，与えられたテキスト T のすべての部分文字列を線形領域で表す，よく知られたデータ構造である．接尾辞木を用いることで，長さ m のクエリ P の出現頻度を，T の長さに依存せず $O(m \cdot \log |\Sigma|)$ 時間で求めることができる．ここに，$|\Sigma|$ はアルファベットの大きさである．

しかし，標準的な実装の場合接尾辞木はテキスト長の 30 倍程度の主記憶領域を必要とするため，大きなテキストに対しては構築し難いという問題がある．したがって，応用によっては，限られた主記憶領域で索引を構築し，近似を行って真の出現頻度を推定することが現実的である．本研究の目的は，より大きなテキストに対して構築可能な，文字列の出現頻度をできるだけ高精度に推定する索引を提案することである．

与えられたテキスト T に対し，任意のクエリの T 中の出現頻度を推定する手法としては，Krishnan ら [2] が提案した索引構造である頻度刈り込み接尾辞木を用いる手法がある．頻度刈り込み接尾辞木は，接尾辞木から出現頻度が閾値以下である部分文字列に対応するノードをすべて削除することで得られる木構造であり，各ノードには対応する部分文字列の出現頻度を格納する．こうして

残されたノードのみを用いて，出現頻度を推定する文字列 P をいくつかの部分文字列に分割し，それらの個別の出現確率の積として P の出現確率を推定することができる．また，Jagadish らは同じ頻度刈り込み接尾辞木を用いてさらに推定精度を改善させる手法を提案している [1]．ただし，頻度刈り込み接尾辞木は一度接尾辞木を構築することで得られるため，一時的には結局接尾辞木と同等の主記憶領域を必要とするため，実際の大規模テキストに対しては適用が難しい．

動的刈り込み接尾辞木 [5] は，木のノード数を一定以下に保ちつつ，与えられたテキスト T 中で出現頻度の高い文字列はできる限り索引化するためのデータ構造である．動的刈り込み接尾辞木の構築では，ノード数が定められた値 M を超えないならば Ukkonen の接尾辞木構築アルゴリズム [3] に基づいて拡張を行う．そしてノード数が M を超えそうになったとき，出現頻度が最も低い文字列に対応するノードをまとめて削除する．したがって，計算機の主記憶領域に合わせて容易に M を定めることができる．この動的刈り込み接尾辞木と，刈り込まれた接尾辞木を用いて真の部分文字列出現頻度を推定する手法である MO（Maximal Overlap）法 [1] を組み合わせることで，任意の文字列の出現頻度を推定することが可能である．

しかし，入力されたテキスト全体を保持し続けなければならないという問題が依然存在しており，さらなる主記憶領域消費量の削減を必要としていた．本稿では，圧縮トライの再構築という処理を行うことで，元のテキストから必要な部分だけを残した新たなテキストを生成することで，索引構造として要求する主記憶領域を削減するアルゴリズムを提案する．このとき，新たなテキストの長さは最悪でもすべてのラベルの合計長である．なお，再構築以外のアルゴリズムは文献 [5] と共通であり，詳細はそちらを参照されたい．

本稿の構成は以下の通りである．2 章では基本的な定義をする．3 章では動的刈り込み接尾辞木を導入する．4 章では圧縮トライの再構築手法を提案する．5 章では提案する再構築アルゴリズムに関する計算機実験を行った結果について述べる．6 章では本稿の結論を述べる．

2. 準　　備

2.1 基本的な定義

アルファベットを Σ とする．Σ 上の文字列全体の集合を Σ^* で表す．文字列 $x \in \Sigma^*$ の長さを $|x|$ と表す．特に長さが 0 の文字列を**空語**といい，ε で表す．Σ^+ を $\Sigma^* \setminus \{\varepsilon\}$ と定義する．文字列 $x, y \in \Sigma^*$ の連結を $x \cdot y$ で表す．ある文字列 $S \in \Sigma^*$ に対して，$S = xyz$ となる $x, y, z \in \Sigma^*$ が存在するとき，x, y, z をそれぞれ S の**接頭辞**，**部分文字列**，**接尾辞**と呼ぶ．S の i 番目の文字を $w[i]$ と表し，S の i 番目から j 番目までの部分文字列を $S[i..j]$ で表す．$i > j$ のとき，便宜的に $S[i..j] = \varepsilon$ と定義する．

Σ 上の文字列の集合 W が**閉じている**とは，すべての $x \in W$ について，任意の x の部分文字列 $y \in \Sigma^*$ に対し $y \in W$ が成り立つことをいう．

文字列 P が文字列 $T[1..n]$ 中に位置 i で出現するとは，$P = S[i..|P|](1 \leqq i \leqq n - |P|)$ である i が存在することをいう．T 中における P の**出現頻度**とはそのような位置 i の個数であり，$f(T, P)$ と書く．ただし，$P = \varepsilon$ ならば $f(T, P) = |T|$ と定義する．以降では，このような文字列 T, P をそれぞれ**テキスト**，**クエリ**と呼ぶ．

本稿で取り組む問題は以下の通りである．

部分文字列出現頻度推定問題: 与えられたテキスト $T \in \Sigma^*$ に対し，任意のクエリ $P \in \Sigma^*$ の T 中の出現頻度を推定するコンパクトな索引を構築せよ．

2.2 接 尾 辞 木

長さ $n \geqq 1$ のテキスト T を，$T[1 \ldots n-1]$ に現れない特別な記号 \$ で終わる文字列と仮定する．T の**接尾辞木**は T のすべての接尾辞を表す圧縮トライ $ST(T) = (V, root, E, suf)$ である．ここで，V はノードの集合である．E は辺の集合で，$E \subseteqq V^2$ である．ノード $s, t \in V$ について，$(s, t) \in E$ のとき，s を t の**親**，t を s の**子**と呼ぶ．t の親を $parent(t)$ で表す．また，子を持たないノードを**葉**と呼ぶ．$root$ は木 $ST(T)$ の根である．$root$ からノード $s \in V$ へのパス上にあるノード $t \in V$ を s の**祖先**，s を t の**子孫**と呼ぶ．s は s 自身の祖先かつ子孫であることに注意する．任意の辺 $e \in E$ には T の空でない部分文字列 $T[j..k]$（$1 \leqq j \leqq k \leqq |T|$）がラベル付けされる．ラベルを $label(e)$，位置の組を $pos(e)$ と書く。実際にはラベルは位置の組 j, k で表される。すなわち，$pos(e) = (j, k)$ ならば $label(e) = T[j..k]$ である．辺が向かうノードはただ一つであるから，$t \in V$ へ向かう辺のラベルも簡単のため $label(t)$ と書くことにする．便宜上，$label(root) = \varepsilon$ と定義する．任意のノード $s \in V$ について，**ノード s が表す文字列**とは，$root$ から s への祖先の列 $root, a_1, a_2, \ldots, s$ の各ラベル文字列を連結した文字列 $label(root) \cdot label(a_1) \cdot label(a_2) \cdots label(s)$ であり，$[s]$ と書く．また，文字列 x を表すノードを $\overline{x}$ と書く．

suf は V 上の関数 $suf : V \to V$ であり，ノード $s, t \in V$ と文字 $a \in \Sigma$ に対して，$[s] = a \cdot [t]$ であるとき，$suf(s) = t$ である．任意の実ノード $s \in V$ について，$[s] = a \cdot [t]$ となる実ノード $t \in V$ が存在する [3]．この関数は s から t への辺 (s, t) とみなすこともできるので，これを**接尾辞リンク**と呼ぶ．

3. 動的刈り込み接尾辞木

動的刈り込み接尾辞木の構築手続きを図 1 に示す．動作は以下の通りである．まず整数 $M > 0$ は**最大ノード数**であり，これを超えないように木を管理する．ノード数が M 未満である限りは，Ukkonen [3] による接尾辞木のオンライン構築アルゴリズムと同様の更新手続き **update** を行う．時点 i において ϕ は 2 回以上 $T[1 \ldots i-1]$ に出現する最長の接尾辞を表すノードの参照である．ある時点 i で **update** を行ったときにノード数が M を超えた場合，更新の前に手続き **prune** を行い，ノード数を削減してから木を拡張する．削除するノードを決定するため，各ノードに**カウンタ** $c(s)$ を持たせる．カウンタの初期値は，葉の場合は 1 とし，それ以外は 0 とする．木の刈り込み手続き **prune** が一度も行われなかったとき，各ノード s について $[s]$ の T 中の出現頻度 $f(T,[s])$ は s を根とする部分木の葉の数と等しく，また接尾辞木と同様に s を根とする部分木のカウンタの合計と等しい．以降ではこの値を s の**合計頻度**と呼び，$C(s)$ で表す．一度の **prune** によって削除されるノードは，カウンタの値が 1 であるノードすべてである．このとき，削除されなかったノードのカウンタも一律に 1 減らす．

内部ノード s については，合計頻度 $C(s)$ が 1 だけ減るよう調整する．すべての葉は 1 だけカウンタを減らされるから，そのままでは $C(s)$ は s の部分木の葉の数だけ減ることになる．ここで，s のすべての子 t がそれぞれ正しく 1 だけ $C(t)$ を減らされたと仮定すると，(子の数 -1) を s に足すことによって，$C(s)$ を 1 だけ減らすことができる．

このようにして構築した動的刈り込み接尾辞木に，閉じた部分文字列集合を表す索引を用いた文字列出現頻度推定手法である [1] や [2] を適用することで，任意の文字列の T 中の出現頻度を推定することが可能である．

刈り込みの後に行われる手続き **reconstruct** は，ラベルと参照文字列を作り直すことで，参照文字列が無限に長くなることを防ぐ手続きである．次節では具体的な 3 つのアルゴリズムを提案する．

手続き **construct-**$DPST$ (T, M);
入力: 長さ n の文字列 T，最大ノード数 M;
出力: 動的刈り込み接尾辞木 $DPST(T, M)$;

```
1  DPST = 根だけからなる木;
2  φ = ε;
3  for i = 1 ... n
4      update(DPST, T[i], φ);
5      if 現在のノード数 > M
6          (DPST, φ)=prune(DPST, φ);
7          reconstruct(DPST);
8  return DPST;
```

図 1　動的刈り込み接尾辞木の構築手続き

手続き **update**
入力: $DPST(T[1 \ldots i-1], M)$, ϕ, $T[i]$;
出力: $DPST(T[1 \ldots i], M)$,ϕ;

```
1  r = root;
2  while φ·T[i] を表すノードが DPST に存在しない
3      if φ̄ が存在しない
4          φ を表すノード s を作る;
7      φ̄ の子 t，辺 (φ̄, t)，ラベル label(t) = (i, ∞) を作る;
8      c(t) = 1;
9      if r ≠ root
10         suf(φ̄) = r;
11     r = φ̄;
12     φ = φ の 1 文字短い接尾辞;
13 if r ≠ root
14     suf(φ̄) = r;
15 φ = φ·T[i];
16 return (DPST, φ);
```

図 2　動的刈り込み接尾辞木の更新手続き

4. 圧縮トライの再構築

ノードの集合 V，辺の集合 E，E の各ノードのラベルが参照する文字列 $S \in \Sigma^*$ からなるパス圧縮トライを $CTrie(V, E, S)$ と表す．S を $CTrie(V, E, S)$ の**参照文字列**と呼ぶ．

ここで閉じた部分文字列集合を表すパス圧縮トライ $CTrie(V, E, S)$ について考える．いま，新たな参照文字列を $S' \in \Sigma^*$ とする．$CTrie(V, E, S)$ のすべての辺 $e \in E$ に対し $S'[i..j] = label(e)$ である区間 (i, j) が存在するならば，全ての辺のラベルが S' を参照するよう変更することで，$CTrie(V, E, S)$ と同じ部分文字列集合を表す圧縮トライ $CTrie(V, E, S')$ を構築することが可能である．このように，同じ部分文字列集合を表すよう維持しながら新たな参照文字列を用いるよう圧縮トライを構成することを，圧縮トライの**再構築**と呼ぶ．以下では 3 つの再構築アルゴリズムを与える．

4.1 自明なアルゴリズム

図 4 に自明なアルゴリズム **reconstruct-naive** を示す．**reconstruct-naive** は，すべての辺 $e \in E$ について $label(e)$ を参照文字列の末尾に追加する．新たに追加した文字列を e が参照するようラベルを変更することで，圧縮トライは明らかに正しく再構築される．

手続き prune
入力: $DPST(T[1\ldots i-1], M)$, ϕ, δ;
出力: 刈り込まれた $DPST(T[1\ldots i-1], M)$;
  while $C(\overline{\phi}) == \delta$
    if $\overline{\phi}$ が存在しない
      ϕ を表すノード s を作る;
    $c(\overline{\phi}) = c(\overline{\phi}) + 1$;
    $\phi = \phi$ の 1 文字短い接尾辞;
  $DPST$ を深さ優先探索し，各ノード s に以下を行う:
    if $c(s) == 1$
      delete u;
    else
      $c(s) = c(s) + (s$ の子の数 $- 1)$;
  return $DPST$;

図 3　動的刈り込み接尾辞木の刈り込み手続き

4.2　接尾辞リンクを用いたアルゴリズム

図 5 に，接尾辞リンクを用いてより短い参照文字列を生成するアルゴリズム **reconstruct-suf** を示す．

閉じた部分文字列集合を表す圧縮トライ $CTrie(V, E, S')$ に対し，ノードの部分集合 $R \subseteqq V$ を $R = \{r \in V | r$ は葉かつ $suf(u) = r$ となるノード $u \in V$ が存在しない $\}$ とおく．ある文字列 $x \in R$ を表すノード $u \in V$ に対し，u の接尾辞リンクをたどって得られるノードの列 $suf(u), suf(suf(u)), \ldots$ はすべて $[u]$ の接尾辞を表すから，$[u]$ が S に存在するならば，これらのノードはすべて u と同じ位置で終わる S の部分文字列を参照すればよいことがわかる．また，u の先祖 $parent(u), parent(parent(u)), \ldots$ についても同様に，そのラベルを追加した上で接尾辞リンク先のノードも同じ部分文字列を参照することで，すべてのノードの参照先を更新することができる．

4.3　参照文字列の再利用によるアルゴリズム

自明なアルゴリズム **reconstruct-naive** では，すべての辺のラベルを新たな参照文字列 S に追加するため，同じ文字列が重複して S に存在することになる．これを改善するのが，図 6 に示す参照文字列にまだ存在しない部分文字列のみを新たに追加するアルゴリズム **reconstruct-search** である．**reconstruct-search** では，各辺 $e \in E$ に対し新たに生成する途中である参照文字列 S' から $label(e)$ を検索し，存在するならばその位置を新たな e の参照位置とし，存在しないならば S の末尾に $label(e)$ を追加した上で e のラベルとして参照する．このとき S の接尾辞木を構築し逐次更新することで，検索を効率よく行うことができる．

手続き reconstruct-naive
入力: 圧縮トライ $CTrie(V, E, S)$;
出力: 再構築された圧縮トライ $CTrie(V, E, S')$;
  $S' = \varepsilon$;
  for each $e \in E$
    $S' = S' \cdot label(e)$;
    $pos(e) = (|S'| - |label(e)| + 1, |S'|)$;
  return $CTrie(V, E, S')$;

図 4　圧縮トライの再構築を行う自明なアルゴリズム

手続き reconstruct-suf
入力: 圧縮トライ $CTrie(V, E, S)$;
出力: 再構築された圧縮トライ $CTrie(V, E, S')$;
  $R = \{r \in V | r$ は葉かつ $suf(u) = r$ となるノード $u \in V$ が存在しない $\}$;
  $S' = \varepsilon$;
  for each $r \in R$
    $a = r$;
    while a が未更新
      $S' = S' \cdot label(a)$;
      $s = a$;
      while s が未更新
        $j = |S'|$;
        $u = s$;
        while u が未更新
          $pos(s) = (j - |label(u)| + 1, j)$;
          $j = j - |label(u)|$;
          u を更新済みとする;
          $u = parent(u)$;
        $s = suf(s)$;
      $a = parent(a)$;
  return $CTrie(V, E, S')$;

図 5　接尾辞の関係を用いた再構築アルゴリズム

5.　計算機実験

本節では，動的刈り込み接尾辞木における 3 つの再構築アルゴリズム **reconstruct-naive**，**reconstruct-search**，**reconstruct-suf** を実装し，実際のテキストデータに対して行った実験結果について述べる．ただし，**reconstruct-search** は辺を選ぶ順番によって結果が大きく変化するため，深さ優先探索による通りがけ順と辺の長い順の 2 つの方法を用いた．以下では表記の簡潔のため，**reconstruct-naive** を **naive**，**reconstruct-suf** を **suf**，深さ優先探索による通りがけ順の **reconstruct-**

手続き reconstruct-search
入力: 圧縮トライ $CTrie(V, E, S)$;
出力: 再構築された圧縮トライ $CTrie(V, E, S')$;
  $S' = \varepsilon$;
  for each $e \in E$
    if $S'[i..j] = label(e)$ である位置 (i, j) が存在する **then**
      $pos(e) = (i, j)$;
    else
      $S' = S' \cdot label(e)$;
      $pos(e) = (|S'| - |label(e)| + 1, |S'|)$;
  return $CTrie(V, E, S')$;

図 6　参照文字列の再利用を行う再構築アルゴリズム

search を **inoder**，辺の長い順の **reconstruct-search** を **sort** と書く．

5.1 データ

テキストデータとして毎日新聞コーパスを用いた．まず 1991 年の全データから本文のみを抽出して連結した文字列 T を作成した．文字列長は $|T| = 38714284$ であった．

5.2 方法

文字列 T から動的刈り込み接尾辞木を構築する際，4 つの再構築アルゴリズムにより生成された参照文字列の長さを刈り込みの度に求め，その平均値をとった．最大ノード数は $10000 \leqq M \leqq 10000000$ まで変化させて実験を行った．

5.3 結果

結果を図 7 に示す．ただし **naive** は多手法の 10 倍以上の参照文字列を生成したため，グラフへの掲載は $M = 500000$ までとした．

naive の結果が芳しくなかったのは，今回用いたデータでは文単位での長い共通な文字列が存在しており，長いラベルが多く存在したことが原因として考えられる．この結果は参照文字列が無視できないメモリ領域を消費することを示しており，実用的に十分であるとは言い難い．

その他の 3 つのアルゴリズムでは，**sort** が他手法を大きく凌ぐ結果を示した．長いラベルを先に参照文字列に追加するため，そのすべての部分文字列が同じ位置を参照できることが大きな効果を生んだと考えられる．また，**inoder** ではノード数の増加に対し参照文字列長の増加が比較的大きかったため，ノード数が多いときには **suf** の方がよい結果を示した．ただしこの 3 つのアルゴリズムでは，いずれも参照文字列長がノード数より短くなっており，十分効率よく参照文字列を生成できたことがわかる．

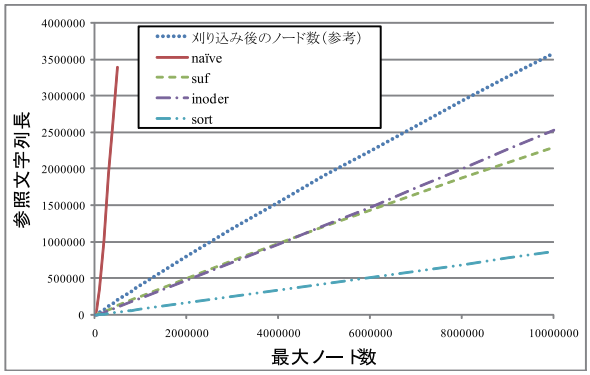

図 7　各再構築アルゴリズムにおける参照文字列長の比較

6. おわりに

本稿では，大規模な文字列が与えられたときに任意の文字列の出現頻度を推定するためのコンパクトな索引である動的刈り込み接尾辞木における，参照文字列の再構築手法を提案した．また計算機実験により，接尾辞リンクの関係を用いた手法や，参照文字列内の検索を行う手法により，生成される参照文字列長を短縮できることを確認した．

今後の課題としては，圧縮トライの再構築を一括ではなく逐次的に行う手法の検討があげられる．また，文字列出現頻度の推定誤差を理論的に明らかにすることが必要である．

文献

[1] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 249–260, New York, NY, USA, 1999. ACM.

[2] P. Krishnan, J. S. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pp. 282–293, New York, NY, USA, 1996. ACM.

[3] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[4] P. Weiner. Linear pattern matching algorithms. In *FOCS*, pp. 1–11, 1973.

[5] 上村卓史, 喜田拓也, 有村博紀. 大規模なテキストに対する部分文字列出現頻度の推定. DEWS2008(電子情報通信学会第 19 回データ工学ワークショップ/第 6 回 DBSJ 年次大会), pp. E7–1, 3 月 2008.