DEWS2007 A2-7

プロパティ付き接尾辞木の構築における境界発見アルゴリズム

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 高度なテキスト検索処理においては，テキストの特性 (プロパティ) を考慮し，ある条件を満たす区間のみを対象として検索を行うという要求が存在する．本論文では，このプロパティを考慮した索引構造を構築する問題に取り組む．2006 年に Amir らは，プロパティに属する区間に含まれる部分文字列のみを格納するプロパティ付き接尾辞木を提案した．彼らの構築アルゴリズムは，接尾辞木から不要なノードを削除することでプロパティ付き接尾辞木を得るものである．本論文では，必要な部分と不要な部分との境界となる位置を見つけるための新たなアルゴリズムを提案する．

キーワード プロパティ付き文字列，文字列照合，接尾辞木．

Finding Borders in Construction of Property Suffix Trees

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 some intelligent application of text retrieval, it is required to do a search just through particular parts of target text data with consideration for some properties that the text have. Namely, the texts are followed by additional information, and the each target part satisfies a certain condition on it. In this paper, we address the problem to construct an efficient index structure for this kind of search. In 2006, Amir *et al.* proposed a modified suffix tree for this problem, called the property suffix tree , which stores only substrings that belong to the given property. This algorithm constructs a suffix tree at first, and then pluning edges indicating substrings which are not included in the property. In this paper, we present an algorithm for finding all borders between nodes to be remained and nodes to be eliminated in the suffix tree.

1. はじめに

文字列照合問題とは，与えられた 2 つの文字列テキスト T とパターン P に対し，T 中での P の出現を求める問題である．この問題は計算機科学において古くから知られていると同時に，インターネット上をはじめとしたさまざまな場所において巨大なデータを扱う上で，現在も非常に重要である．

文字列照合問題を効率的に解くための問題として，与えられた文字列に対し前処理を行うことで複数の文字列照合問題に高速に答えられるようにするという文字列索引問題がある．接尾辞木 [6], [8] は長さ n の文字列 T の全ての部分文字列を表す索引である．接尾辞木を用いれば，長さ m のパターン P に対し，文字列照合問題を $O(m \log |\Sigma| + occ)$ 時間で解くことができる．ここで，$|\Sigma|$ はアルファベットの大きさ，occ は T 中での P の出現回数である．$|\Sigma|$ が定数のときは，接尾辞木は $O(n)$ 時間領域で構築することができる．さらに，Ukkonen は接尾辞木をオンラインで構築するアルゴリズムを示した [7]．また，単語の先頭から始まる文字列のみを扱う索引 [3], [5] や，扱う文字列の長さを限定した索引 [4], [9] など，さまざまな変種が提案されている．

さらに高度な問題として，文字列の特性 (プロパティと呼ばれる) を考慮し，照合すべき部分に制限を加えたプロパティ付き文字列照合問題がある．プロパティは区間の集合 $\{(s_1, f_1), \ldots, (s_k, f_k)\}$ で与えられ，この中の区間に含まれる文字列のみを照合の対象とする問題である (図 1)．プロパティ付き文字列照合問題自体は，文字列照合問題で用いられるアルゴリズムの簡単な拡張で解決できる．しかし，これをより高速に解くための索引構造の構築は未解決だった．近年，Amir ら [1] は接尾辞木を応用し，プロパティの付いた文字列に対する索引構造であるプロパティ付き接尾辞木を提案した．プロパティ付き接尾辞木を用いると，長さ m のパターン P に対し，$O(m \log |\Sigma| + tocc)$ 時間でプロパティ付き文字列照合問題を解くことができる．ここで，$tocc$ は π の区間内における P の出現回数である．

プロパティ付き接尾辞木は，接尾辞木を構築したあと，接尾辞木から不要な枝を刈り込むことで作られる．枝を刈り込む際には，どこまでを削除するかの境界を決める必要がある．この境界は，文字列 $T = t_1 \cdots t_n$ の全ての位置 $1 \leqq i \leqq n$ について，i 番目から始まる文字列がどこまでプロパティの区間内に含まれるかを表す位置 $end(i)$ で表される．つまり，全ての位置 $1 \leqq i \leqq n$ について，$t_i \cdots t_{end(i)}$ という文字列を表す接尾辞木上の位置が境界となり，この境界より下にある枝を刈ればよい．

境界を求める自明なアルゴリズムとして，位置 i について文字列 $t_i \cdots t_{end(i)}$ を木の根から探索する方法が考えられる．しかし，T 全体がプロパティの区間に含まれる場合，すなわち $end(1) = end(2) = \cdots = end(n) = n$ のとき，この方法では明らかに $O(n^2)$ 時間かかってしまう．これに対し，Amir らは全ての境界をを $O(n \log \log n)$ 時間で求める方法を示した．

本論文では先の自明なアルゴリズムの考え方を元に，ある文字列 $t_i \cdots t_j$ を表すノードから $t_{i+1} \cdots t_j$ を表すノードへのリンクである接尾辞リンクを利用した新しいアルゴリズムを提案する．このアルゴリズムでは，$t_{i-1} \cdots t_{end(i-1)}$ を表す位置から $t_i \cdots t_{end(i)}$ を表す位置を算出することで，毎回根から探索することなく全ての境界位置を順番に見つけていくことができる．

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 $x \in \Sigma^*$ について，x 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．

文字列 $x_1, x_2 \in \Sigma^*$ の連結を $x_1 \cdot x_2$ で表す．ただし，特に混乱が生じないかぎりはこれを省略し，$x_1 x_2$ と表す．

ある文字列 $w \in \Sigma^*$ に対して，$w = xyz$ となる

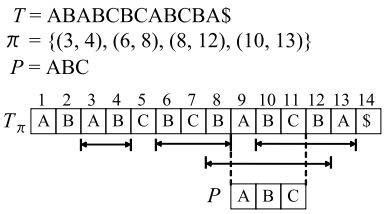

図 1　プロパティ付き文字列照合の例

$x, y, z \in \Sigma^*$ が存在するとき，x, y, z をそれぞれ w の**接頭辞**，**部分文字列**，**接尾辞**と呼ぶ．w の i 番目の文字を $w[i]$ と表し，w の i 番目から j 番目までの部分文字列を $w[i \ldots j]$ で表す．$i > j$ のとき，$w[i \ldots j] = \varepsilon$ と定義する．文字列 T の i 番目の接尾辞 $T[i \ldots |T|]$ を T^i と表す．

2.2 文字列のプロパティ

文字列 T が何らかの特性を持つ場合を考える．**文字列 T のプロパティ π** を，区間の集合 $\pi = \{(s_1, f_1), \ldots, (s_t, f_t)\}$ で表す．π の要素数を $|\pi|$ で表す．$1 \leqq i \leqq t$ について，$s_i, f_i \in \{1, \ldots, n\}$，$s_i \leqq f_i$ が成り立つ．

プロパティ π を持った文字列 T を**プロパティ付き文字列**といい，T_π で表す．ここで，長さ n の文字列 T のプロパティ π が以下の性質を持つとき，π は**標準形**であるという．

(1)π の全ての区間が明示的に与えられる．

(2)$1 \leqq i \leqq n$ について $s_k = i$ となるような区間 $(s_k, f_k) \in \pi$ が高々ひとつ存在する．

(3)$s_1 < s_2 < \cdots < s_{|\pi|}$.

つまり，π が標準形であるならば，全ての区間が開始位置の昇順で与えられ，それらの開始位置は全て異なる．簡単のため，以降はこの標準形のプロパティのみを扱う．

3. プロパティ付き文字列索引問題

3.1 プロパティ付き文字列照合問題

プロパティ付き文字列 T_π とパターン P に対し，プロパティ付き文字列照合問題とは以下のような問題である．

プロパティ付き文字列照合問題：プロパティ付き文字列 T_π とパターン P が与えられたとき，$P = T[i \ldots j]$ であり，$s_k \leqq i$ かつ $j \leqq f_k$ となる $(s_k, f_k) \in \pi$ が存在する全ての位置 i を求めよ．

詳しい説明は省略するが，この問題は既存の文字列照合アルゴリズムを適用し，π 中の区間に含まれているか否かを検査することで容易に解くことができる．

3.2 プロパティ付き文字列索引問題

プロパティ付き文字列に対し一度前処理をすることで，より多くの問い合わせを高速に処理したいという要求がある．そこで，次のような問題を考える．

プロパティ付き文字列索引問題：プロパティ付き文字列 T_π が与えられたとき，プロパティ付き文字列照合問題を，パターン P の長さ m と T 中で π に含まれる区間内での

P の出現回数 $tocc$ に対し，$O(m+tocc)$ 時間で答えられるように前処理せよ．

π を考慮せずに，T の全ての部分文字列に対しての文字列索引を構築し，T 中での全ての P の出現位置 $\{(i_1, i_1+m-1), \ldots, (i_k, i_k+m-1)\}$ を求めてからそれぞれが π の区間に含まれるかを求めたのでは，π の区間に含まれない P の出現に対する計算時間がかかるため，真の出力サイズ $tocc$ に比例する時間では答えられない．

4. プロパティ付き文字列索引の構築

4.1 接尾辞木

接尾辞木は長さ n の文字列 T の全ての部分文字列を表現するデータ構造である．以下では文字列 T の最後の文字 $T[n] = \$$ は $T[1 \ldots n-1]$ には含まれない文字（**終端記号**と呼ばれる）とする．文字列 T の接尾辞木 $ST(T)$ は $ST(T) = (V \cup \{\perp\}, root, E, suf)$ の四つ組みで表される．ここで，V はノードの集合であり，$\perp \notin V$ とする．E は辺の集合であり，$E \subseteqq V^2$ である．接尾辞木において，このグラフ (V, E) は $root \in V$ を根とする木を成している．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．$\perp$ は $root$ の親となる特別なノードである．また，子を持つノードを**内部ノード**，子を持たないノードを**葉ノード**と呼ぶことにする．さらに，$root$ 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となる．ノード u を根とする部分木を $ST(u)$ と表す．

辺 $e \in E$ にはそれぞれ空でない**ラベル文字列**が与えられ，T 上の位置の組 (j,k) で表される．すなわち，このときのラベル文字列は $T[j \ldots k]$ である．$e = (s,t)$ とすると，t に向かう辺はただ 1 つだけであるから，t へ向かう辺のラベル文字列を $label(t)$ で表すことにする．任意のノード $s \in V$ について，そのすべての祖先のラベル文字列を $root$ から順に連結したものを $\overline{s}$ と書き，これを**ノード s が表す文字列**と呼ぶ．すなわち，$root, a_1, a_2, \ldots, s$ をノード s の祖先の列とすると，$\overline{s} = label(root) \cdot label(a_1) \cdot label(a_2) \cdots label(s)$ である．ただし，$label(root) = \varepsilon$ と定義する．

接尾辞木 $ST(T)$ 上の各葉ノード全体は T の接尾辞全体と一対一で対応し，$root$ を除く内部ノードは必ず子を二つ以上持つ．このとき，葉ノードは n 個存在し，全体では $2n-1$ 個以下のノードを持つ．T^i を表す葉ノードを $leaf(i)$ と表す．また，内部ノード $s \in V$ の任意の二つの子は互いに異なる文字から始まるラベル文字列を持つ．よって，各ノード $s \in V$ は T のある特定の部分文字列と一対一に対応する．$ST(T)$ 上に対応するノードが存在しないその他の部分文字列については**仮想ノード**として扱われる．これと区別するため，以降は V の要素を**実ノード**と呼ぶ．

```
procedure canonize(v = (s,(k,i)));
method:
  if (k > i)
    return (s,(k,i));
  s'=T[k] で始まるラベル文字列を持つ s の子;
  while (|label(s')| ≦ i − k + 1) {
    (s,(k,i)) = (s', k + |label(s')|, i);
    if (k ≦ i)
      s'=T[k] で始まるラベル文字列を持つ s の子;
  }
  return (s,(k,i));
end.
```

図 2　仮想ノード $v = (s,(k,i))$ を正規形にするアルゴリズム

$ST(T)$ 上の仮想ノード v が表す文字列が $\overline{v} = T[j \ldots i]$ だとすると，v を (j,i) で表すことができる．また，v の実ノードの祖先 s の表す文字列を $\overline{s} = T[j \ldots k-1]$ とすると，$\overline{v} = \overline{s} \cdot T[k \ldots i]$ である．したがって，仮想ノード v を $v = (s,(k,i))$ と表す．ただし，特に必要がない場合，以降は仮想ノード $v = (s,(k,i))$ は単に v と表す．s が v の最も近い実ノードの祖先であるとき，$(s,(k,i))$ は**正規形**であるという．$v = (s,(k,i))$ を正規形にするアルゴリズム $canonize$ を図 2 に示す．v が実ノード s そのものを表すときは，正規形は $(s,(i+1,i))$ となり，実ノードも含めて一般的に表すことができる．

接尾辞木 $ST(T)$ 上の任意のノード $u \in V$ と文字 $a \in \Sigma$ に対し，$\overline{u} \cdot a$ に対応するノードを $child(u,a)$ と表す．$child(u,a)$ は仮想ノードの場合もある．$ST(T)$ 上に対応するノードが無い場合は未定義とする．任意の文字 $a \in \Sigma$ に対し，$child(\perp, a) = root$ と定義する．u が仮想ノード $u = (s,(k,i))$ のときは，$T[i+1] = a$ であるならば，$child(u,a) = (s,(k,i+1))$ である．u から $child(u,a)$ へ移動することを u **から** a **で進む**という．

関数 $suf : V \to V$ は，任意の実ノード $s \in V$ に対して $\overline{s}$ より一文字短い接尾辞を表す実ノードを返す関数である．すなわち，実ノード $s, t \in V$ と文字 $a \in \Sigma$ に対して $\overline{s} = a \cdot \overline{t}$ であるとき，$suf(s) = t$ である．これは実ノード s から t への辺 (s,t) と見ることもでき，この辺を**接尾辞リンク**と呼ぶ．特に $suf(root) = \perp$ とし，$suf(\perp)$ は未定義とする．任意の内部ノード $s \in V$ について，$\overline{s} = a \cdot \overline{t}$ となる内部ノード $t \in V$ が存在する．

仮想ノード v の接尾辞リンクは明示的には表されないが，実ノードの祖先 s の接尾辞リンクを使って仮想的に実現する．$\overline{v} = \overline{s}\alpha$ とすると，v の接尾辞リンク先は，$suf(s)$ から α の各文字で進んだノード u である．すなわち，$v = (s,(k,i))$ とすると，$u = (suf(s),(k,i))$ であ

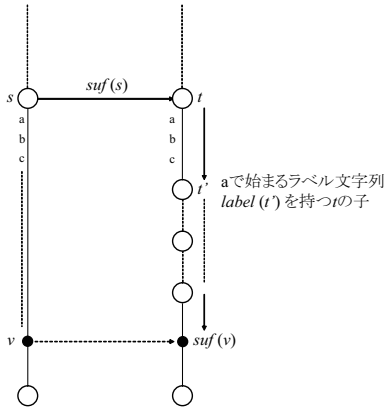

図 3　仮想ノードの接尾辞リンク

T = ABABCBCABCBA$
π = {(3, 4), (6, 8), (8, 12), (10, 13)}

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	A	B	A	B	C	B	C	B	A	B	C	B	A	$
$end(i)$	0	1	4	4	4	8	8	12	12	13	13	13	13	13

T[3...4] A B / B / ε

T[6...8] B C B / C B / ε

T[8...12] B A B C B / A B C B / ε

T[10...13] B C B A / C B A / B A / A / ε

図 4　プロパティに含まれる文字列と $end(i)$ の関係

る．u を正規形 $(s', (k', i))$ にするとき，実際に $suf(s)$ から α の各文字で 1 文字ずつ進む必要はなく，仮想ノードを飛ばして，辺の長さに関係なく実ノードだけをたどっていくことができる．図 3 に例を示す．

4.2　プロパティ付き接尾辞木

Amir ら [1] はプロパティ付き文字列索引問題を解くため，接尾辞木を元にした新しいデータ構造である**プロパティ付き接尾辞木**を提案した．

プロパティ付き文字列 T_π において，π に含まれる T の部分文字列 S があるとき，S の部分文字列もまた π に含まれる．したがって，文字列 T の位置 i から始まる部分文字列 $T[i \ldots j]$ の中で，π に含まれる最も長い文字列の**終了位置** $end(i)$ を，$(s_k, f_k) \in \pi$ かつ $s_k \leqq i \leqq f_k$ であるような f_k の中で最大なものと定義する．もしそのような f_k が存在しないならば，$end(i) = i - 1$ とする．また，$end(0) = 0$ と定義する．$end(i)$ を計算し，プロパティに含まれる部分文字列を表した例を図 4 に示す．π が標準形で与えられているならば，全ての $end(i)$ を $O(n)$ 時間で計算できる．ここで次の補題が成り立つ．

［補題 1］　$end(0) \leqq \cdots \leqq end(n)$.

[証明]　まず $end(1) \geqq 0$ であるから $end(0) \leqq end(1)$ が成り立つ．ある位置 $1 < i \leqq n$ で $end(i-1) > end(i)$ であると仮定すると，$T[i-1 \ldots end(i-1)]$ は $T[i \ldots end(i)]$ を完全に含む．よって $end(i)$ の定義から $end(i) = end(i-1)$ でなければならず，仮定と矛盾する．よって全ての位置で $end(i-1) \leqq end(i)$ である．　□

プロパティ付き文字列 T_π と接尾辞木 $ST(T)$ のノード u が与えられたとき，$leaf(i)$ が $ST(u)$ に含まれており，かつ $end(i_j) - i_j \geqq |\overline{u}|$ となる全ての i_j の集合を $S_\pi(u)$ と表す．ここで u とその親 v について，$end(i_j) - i_j \geqq |\overline{u}| > |\overline{v}|$ であるから，次の補題が成り立つ．

［補題 2］　プロパティ付き文字列 T_π と接尾辞木 $ST(T)$ のノード u, v が与えられ，v が u の親であるならば，$S_\pi(u) \subseteqq S_\pi(v)$ である．

$ST(T)$ の根から $leaf(i)$ への道は，$i \in S_\pi(u)$ となる全ての実ノードからなる道 p_1 と，$i \notin S_\pi(u)$ となる全ての実ノードからなる道 p_2，p_1 と p_2 をつなぐ辺 e に分けられる．p_1 上で最も深い位置にある実ノードを s とすると，$T[i \ldots end(i)]$ を表す**境界ノード** $border(i)$ は次の 2 つの場合に分けて定義される: $end(i) - i + 1 = |\overline{s}|$ ならば $border(i) = w$，そうでなければ $border(i)$ は辺 e の上の仮想ノードである．また，$border(0) = root$ と定義する．

プロパティ付き文字列 T_π に対し，プロパティ付き接尾辞木 $PST(T_\pi)$ は $PST(T_\pi) = (V' \cup \{\perp\}, root, E', suf)$ で表される．ここで，V' は接尾辞木のノードの集合 V の部分集合である．E' は辺の集合であり，$E' \subseteqq V'^2$ である．$\perp, root, suf$ は $ST(T)$ と同じである．任意の内部ノード $u \in V'$ について $S_\pi(u)$ は空ではない集合である．また，任意の葉ノード $v \in V'$ について $S_\pi(v)$ が空集合ならば，v へ向かう辺 e はある $border(i)$ を e 上の仮想ノードに持つ辺である．

4.3　構築アルゴリズム

Amir ら [1] による $PST(T_\pi)$ の構築アルゴリズムを図 5 に示す．このアルゴリズムでは，最初に接尾辞木 $ST(T)$ を構築する．そして全ての位置 i について $border(i)$ を求め，各実ノード s について $s = border(i)$ となるような全ての位置 i のリスト $list(s)$ を，各辺 e について e 上の仮想ノード v が $v = border(i)$ となるような全ての位置 i のリスト $list(e)$ を作成する．$border(i)$ の計算においては，$end(i) - i \leqq |\overline{u}|$ となる $leaf(i)$ に最も近い祖先 u を $O(\log \log n)$ 時間で求める手法 [2] を用いる．次に，$list(s)$ が空でない実ノード s か，または $list(e)$ が空でない辺 e に接続されている実ノードに印をつける．補題 2

procedure $PST(T_\pi)$;
method:
  $ST(T)$ を作る;
  for $i = 1$ **to** n
    $end(i)$ を計算する;
  for $i = 1$ **to** n
    $leaf(i)$ に対し $loc(i)$ を見つけ，$list$ に i を追加する;
  $S_\pi(u)$ が空である実ノード u と u に接続する辺 e を削除する;
  子を 1 つしか持たないノードを削除し，辺を結合する;
  foreach $e \in E'$
    $list(e)$ の前処理をする;
end.

図 5　プロパティ付き接尾辞木の構築アルゴリズム

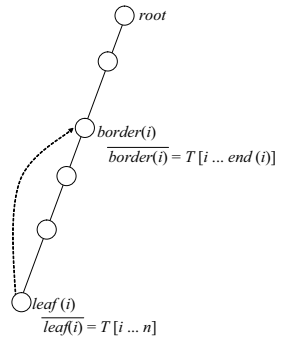

図 6　$leaf(i)$ から $border(i)$ への移動

より，印の付いた実ノード s の親 t について $S_s^\pi \subseteqq S_t^\pi$ であるので，t の祖先にも印をつける．そして，印の付いていない実ノードを削除し，子を 1 つしか持たないノード u も削除する．このとき，$list(u)$ と，u に接続している辺 e_1，e_2 のリスト $list(e_1), list(e_2)$ を結合し，新たな辺 e と $list(e)$ を作成する．最後に，辺のリストの各要素 i に対して，問い合わせに効率的に答えるための前処理を行う．この前処理は次項で説明する．結果として図 7 のようなプロパティ付き接尾辞木が得られる．$PST(T_\pi)$ は，図 6 のように，接尾辞木 $ST(T)$ の $leaf(i)$ を $border(i)$ へ移動することで構築する．

4.4　位置のリストの構築

辺のリストには $end(i) - i$ の値の異なる位置 i が複数含まれているので，長さ m のパターン P が与えられたとき，$end(i) - i \geqq m$ であるような位置 i のみを出力する必要がある．$end(i) - i$ の降順でソートすればこの要求を容易に実現できるが，ソートには多くの計算時間がかかる．そこで Amir らは，k 個の数値の集合の中央値 m が $O(k)$ 時間で求められることを利用し，リストの要素数に比例した計算時間の前処理により，問い合わせに対して効率的に答えることができるようにした．

まず $list(e)$ の全ての要素 i から，$end(i) - i$ の中央値

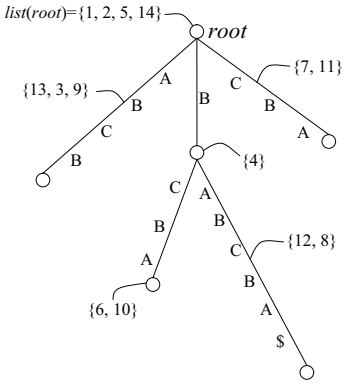

図 7　T=ABABCBCABCBA\$, $\pi = \{(3,4),(6,8),(8,12),(10,13)\}$ に対するプロパティ付き接尾辞木.

m_1 を求める．$end(i) - i > m_1$ であるような全ての要素 i はリストの後半にそのまま配置する．次に $end(i) - i < m_1$ であるような全ての要素 i から，同様に $end(i) - i$ の中央値 m_2 を求め，$end(i) - i > m_2$ であるような全ての要素 i は残りのリストのうちの後半にそのまま配置する．こうして要素数が 1 個になるまで再帰的に分割を繰り返し，要素を配置していく．最終的に $\log m$ 回分割すると，中央値を求めるための計算時間は $O(\Sigma_{i=0}^{\log m} m/2^i) = O(m)$ であり，要素数に対し線形時間で前処理可能である．

4.5　問い合わせ処理

先に述べた前処理を行うと，処理とは逆順に小さく分割された領域から前方に並び，かつそれぞれの領域は中央値以上か以下かで分割される．問い合わせの際は，先頭の領域の要素からパターン P の長さ $|P|$ と $end(i) - i$ の値を比較し，全て出力したら次の領域に進む，という作業を繰り返せばよい．領域内に 1 つでも $|P|$ 以下の値を持つ要素があれば，次以降の領域の要素は全て $|P|$ 以下の値しか持たないので，その領域までで出力は終了である．ある領域で出力を終了したら，その領域より前にある全ての領域の要素は出力しているので，比較した要素のうち少なくとも半分は実際に出力していることになる．よって，出力の個数に比例した時間で問い合わせを処理することができる．

5.　提案アルゴリズム

Amir らの手法では $leaf(i)$ の祖先として $border(i)$ を求めていた．本節では，$border(i-1)$ から接尾辞リンクを用いて移動することで，毎回根からたどることなく $border(i)$ を順に見つけていくアルゴリズムを提案する．

5.1　基本的なアイディア

プロパティ付き文字列 T_π と接尾辞木 $ST(T)$ について，$border(1), \ldots, border(n)$ を順番に求める方法を考える．いま $border(i-1)$ が与えられたとすると，$border(i)$ への移動は次の補題に基づいて行うことができる．

procedure $find\text{-}border(ST(T), \{end(1), \ldots, end(n)\})$;
method:
1 $w = (s, (k, j)) = (root, (1, 0))$;
2 **for** $i = 1$ **to** n {
3　$w = (s, (k, j)) = canonize(suf(s), (k, end(i)))$;
4　w の位置する実ノードまたは辺の $list$ に i を追加する;
5 }
end.

図 8　全ての $border(i)$ を見つけるアルゴリズム

［補題 3］　接尾辞木 $ST(T)$ と $end(1), \ldots, end(n)$ が与えられたとき，$1 < i \leqq n$ において，$border(i)$ は $border(i-1)$ の接尾辞リンク先のノードから $T[end(i-1)+1], \ldots, T[end(i)]$ の各文字で進んだところにあるノードである．

[証明]　$\overline{border(i-1)} = T[i-1 \ldots end(i-1)]$ であるので，$border(i-1)$ の接尾辞リンク先のノードを v とすると，$\overline{v} = T[i \ldots end(i-1)]$ である．補題 1 より $end(i-1) \leqq end(i)$ であり，$\overline{border(i)} = T[i \ldots end(i)]$ であるので，v は $border(i)$ の $end(i) - end(i-1)$ 文字短い接尾辞を表す先祖である．v から $T[end(i-1)+1]$,…,$T[end(i)]$ で進んだ先のノードが表す文字列は $\overline{v} \cdot T[end(i-1)+1] \cdots T[end(i)] = T[i \ldots end(i-1)] \cdot T[end(i-1)+1] \cdots T[end(i)] = T[i \ldots end(i)] = \overline{border(i)}$ となる．よって示された．　□

各時点 i において $border(i-1)$ を保持し，接尾辞リンク先のノード v を求めれば，$border(i)$ は $end(i-1)$ と $end(i)$ の差分の文字数をたどることで求められる．

5.2　アルゴリズム

接尾辞木 $ST(T)$ 上で，接尾辞リンクを利用しながら，全ての $border(i)$ を $i = 1 \ldots n$ の順にたどるアルゴリズム $find\text{-}border$ を図 8 に示す．

このアルゴリズムの動作は以下のとおりである．w は $border(i)$ をたどっていく仮想ノード $w = (s, (k, j))$ である．$w = (s, (k, j))$ が正規形でないと，$border(i)$ が実際に位置する実ノードまたは辺がわからないので，w は常に正規形にしておく必要がある．最初は $border(0) = root$ であり，$end(0) = 0$ であるから，$w = (root, (1, 0))$ とする．以降のループでは補題 3 にしたがって $border(i)$ に移動する．$T[k \ldots end(i-1)] \cdot T[end(i-1)+1 \ldots end(i)] = T[k \ldots end(i)]$ であるから，v から $T[end(i-1)+1 \ldots end(i)]$ の各文字で進むと，$border(i) = (suf(s), (k, end(i)))$ となる．これを正規形にすると $border(i) = canonize(suf(s), (k, end(i)))$ となるので，求められた境界ノードを保持することで次の境界ノードに進むことができる．

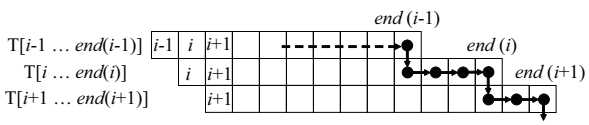

図 9　w の文字列上での動き

w の接尾辞木上の動きを文字列上で表した例を図 9 に示す．接尾辞リンク先への遷移には下への移動が，文字列 $T[end(i-1)+1 \ldots end(i)]$ で進む分には右への移動が対応している．$w = (s, (k, j))$ から接尾辞リンク先のノード $w' = (s', (k', j))$ へ移動する際，図 3 で表したように，$canonize$ は $suf(s)$ から実ノードのみをたどって移動している．これにより，辺のラベル文字列の長さに依らず接尾辞木上を進むことができる．

補題 3 を用いて，以下の定理が得られる．

［定理 1］　長さ n の文字列 T の接尾辞木 $ST(T)$ と終了位置の集合 $\{end(0), \ldots, end(n)\}$ が与えられたとき，アルゴリズム $find\text{-}border$ は $1 \leqq i \leqq n$ について $border(i)$ を正しく計算する．

[証明]　i についての数学的帰納法で証明する．$i = 0$ において $border(0) = root$ であり，定義より明らかに正しい．また補題 3 より，$1 < i \leqq n$ において $border(i-1)$ が計算されていれば $border(i)$ が計算できる．以上より $1 \leqq i \leqq n$ において $border(i)$ が正しく見つけられることが示された．　□

6.　計算機実験

提案アルゴリズムを実装し，人工データについて実験を行った．

6.1　デ　ー　タ

テストデータとして，$\Sigma = \{A, G, C, T\}$ をアルファベットとして持つ人工データを用いた．テキストの各文字列はランダムに発生させた．ただし，実験によって各文字の確率を変化させている．

6.2　実　　験

境界発見アルゴリズムとして，提案手法である suf のほか，比較のため，各境界ノードを毎回根から探索する $root$，毎回葉ノードから探索する $leaf$ を実装した．

実験 1: 全ての文字が等確率に出現するテキストについて，テキスト長 n を変化させて，境界ノードを求める部分の実行時間を測定した．ここで，プロパティ π は，$end(i) = i + 10$ となるように設定した．

実験 2: 各文字の出現確率を $P(A) = P(T) = 0.4$, $P(G) = P(C) = 0.1$ とし，それ以外は実験 1 と同様に実行時間を測定した．

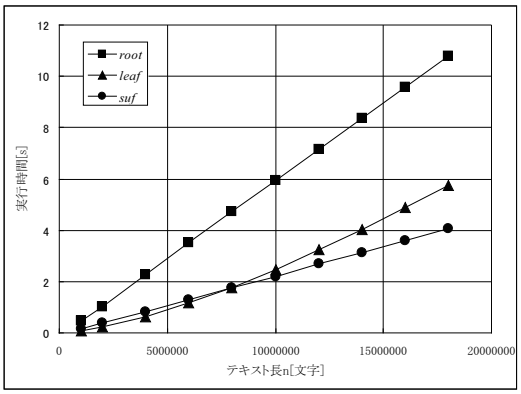

図 10　実験 1:全ての文字が等確率に現れる文字列に対する実行時間の測定．境界ノードの深さは 10 とした．

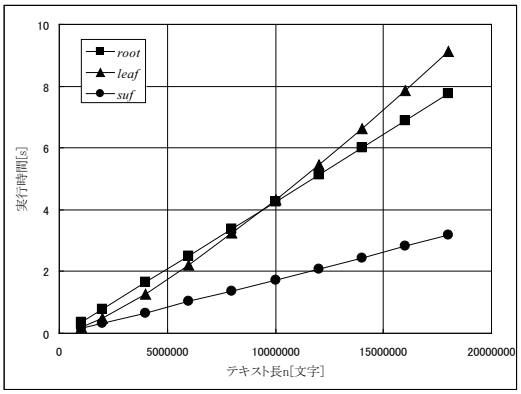

図 11　出現確率に偏りがある文字列に対する実行時間の測定

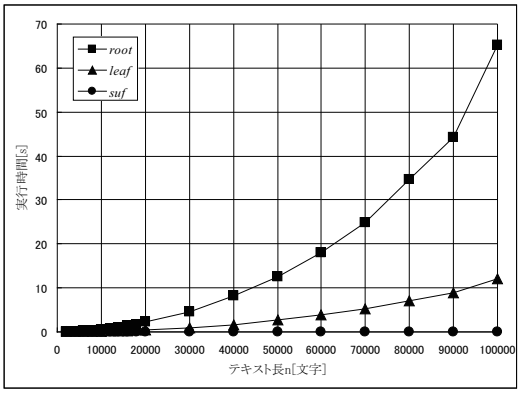

図 12　T =AAA$\cdots$A\$ に対する実行時間の測定．境界ノードは根と葉の中間にある．

実験 3: $P(A) = 1$，つまり T =AAA$\cdots$A\$ とし，さらに，境界ノードが根と葉ノードの中間に位置するように π を与え，実行時間を測定した．

6.3 結　　果

結果 1: 実行結果を図 10 に示す．テキスト長が短いときは $leaf$ が最も高速であるが，長いときには suf のほうが速くなっている．

結果 2: 実行結果を図 11 に示す．$root$，suf は実験 1 とほぼ同じ速度で動作したが，$leaf$ は遅くなっている．

結果 3: 実行結果を図 12 に示す．$root$ と $leaf$ では実行時間が増えたのに対し，suf ではほぼ同じ速度で実行された．これは，実ノードの深さが最大で n となるため，根からも葉からも毎回 $O(n)$ 個の実ノードをたどって境界ノードを見つける必要があるためであると考えられる．

7. おわりに

本論文では，プロパティ付き接尾辞木の構築において接尾辞木から不要なノードを削除する際，その境界となる全ての位置を求めるためのアルゴリズムを示した．

今後の課題としては，まず計算量が未証明である．また，接尾辞木を作らずに，与えられたプロパティ付き文字列からプロパティ付き接尾辞木を直接構築する手法が望まれる．応用としては，1 つの文字列について複数のプロパティが付随し，それぞれのプロパティに合わせた文字列照合を行う問題が考えられる．これに対する索引の構築手法についても検討している．

文　　献

[1] Amihood Amir, Eran Chencinski, Costas Iliopoulos, Tsvi Kopelowitz, and Hui Zhang. Property Maching and Weighted Matching, In Proc. of the 17th Annual Symposium on Combinatorial Pattern Matching , 2006.

[2] A.Amir, D.Keselman, G.M.Landau, M.Lewenstein, N.Lewenstein, and M.Rodeh. Text Indexing and Dictionary Matching with One Error. J.Algorithms, 37(2):309-325, 2000.

[3] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. Algorithmica, 23(3):246.260,1999.

[4] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. Theoretical Computer Science, 304:87.101, 2003.

[5] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In In proc. of 17th Ann. Symp. on Combinatorial Pattern Matching, 2006. (to appear).

[6] E. M. McCreight. A space-economical suffix tree construction algorithm. J. ACM, 23:262.272, 1976.

[7] E. Ukkonen. On-line Construction of Suffix Trees, *Algorithmica*, 14(3):249-260, 1995.

[8] P.Weiner. Linear pattern matching algorithms, Proc. IEEE 14th Annual Symposium on Switching and Automata Theory, pp.1-11, 1973.

[9] 上村卓史，喜田拓也，有村博紀． 単語幅を制約した接尾辞木の効率のよい構築アルゴリズム． 第 5 回情報科学技術フォーラム講演論文集 (FIT2006)，2006.