近似文字列照合による全文検索のための接尾辞配列の高速走査法

内山将夫† 井佐原均†

近似文字列照合による全文検索では，入力パターンと一定以下の編集距離にある部分テキスト全てをテキストから検索する．近似文字列照合による全文検索は，テキストを接尾辞トライにより索引付けし，それを利用して検索することにより実現できる．しかし，接尾辞トライの占める空間領域は大きいため，接尾辞配列を索引として利用することもある．接尾辞配列を索引として利用する場合には，従来研究では，接尾辞トライ上での探索を接尾辞配列上での2分探索により模擬している．それに対して，本稿では，2分探索ではなく，補助的な配列を用いることにより，高速に，接尾辞トライ上での探索を模擬することができる手法を提案した．更に，2分探索による方法を利用した場合と提案手法を利用した場合とにおける検索速度を実験的に測定し，提案手法の方が検索速度が速いことを示した．

Fast Traversal of Suffix Arrays for Full-Text Approximate String Matching

Masao Utiyama† and Hitoshi Isahara†

Given a text and an input pattern, the goal of *full-text approximate string matching* is to search for all parts of the text that match the pattern. Full-text approximate string matching can be performed using a suffix trie as an index of the text. A suffix trie, however, is relatively large. So, a suffix array, which is a compact representation of a suffix trie, is often used to simulate searches on a suffix trie. A binary search algorithm is used to search the array. A method is described in this paper that uses an auxiliary array to simulate searches on a suffix trie. The method does not use a binary search algorithm so that it can perform a faster simulation. Experiments showed that the proposed method is faster than one using a binary search algorithm.

1. はじめに

近似文字列照合による全文検索とは，入力パターンと検索対象テキストとが与えられたとき，テキスト中における部分テキストのうちで，入力パターンとの違いが一定以下の（入力パターンと類似した）部分テキスト全てを検索することを言う．

近似文字列照合による全文検索には，様々な用途がある．たとえば，光学的文字読み取り装置（optical character reader, OCR）により読み取られたテキストを検索したいときには，そのテキストの中には読み取り誤りがある可能性があるので，検索もれをなくすためには，パターンに厳密に一致する部分テキストだけでなく，ある程度の違いも許した部分テキストも検索する必要がある．また，本稿の著者らが主な研究領域としている自然言語処理においても，翻訳メモリ（translation memory）☆などでは，入力文と類似した文（類似用例）を検索する必要があるので，近似文字列照合の技術が役立つ．

近似文字列照合による全文検索の方法には，あらかじめテキストに索引付けをしておいて，その索引を利用してパターンを検索する方法と，索引は利用せずに，パターンが与えられる毎にテキスト全体を走査する方法とがある☆☆．索引を利用する方法では，3章で述べるように，接尾辞トライ（suffix trie）ある

† 通信総合研究所
Communications Research Laboratory

☆ 翻訳メモリとは，対訳文からなるデータベースのことである．たとえば，翻訳メモリとして日英対訳データベースがあり，それを日英翻訳支援に使う場合には，ユーザが，ある日本語文をシステムに入力したときには，システムは，それに類似した日本語文を翻訳メモリから探し，その対訳英文を出力する．ユーザは，出力された英文を参考にして，入力日本語文を翻訳する．

☆☆ 本稿では，索引を利用しない場合については考察の対象外である．なお，索引を利用しない方法についてのサーベイ論文としては，文献11)がある．

いは接尾辞木（suffix tree）[6] を索引とする方法がある[1),2),5),12),13),15)]が，接尾辞トライあるいは接尾辞木は空間領域を多くとるので，その代りに，接尾辞配列（suffix array）[9] を利用することもある[1),2),12)]．接尾辞トライを利用した検索法☆では，トライの根からノードを下に辿ることにより検索をする．また，接尾辞配列を利用した検索では，接尾辞トライにおけるノードの走査を模擬することにより，検索を実現する．

このノードの走査の模擬において，従来の手法[1),2),12)] では，次に走査すべきノードを決めるために2分探索を利用していた．したがって，その分だけ計算コストが掛かっていた．一方，我々が，本稿で提案する方法では，補助的な配列を用いることにより，2分探索をせずに次に走査すべきノードを決めることができる．そのため，2分探索をする方法に比べて，高速に検索することができる．

以下，2章では用語などの準備をし，3章では，接尾辞トライを用いた近似文字列照合による全文検索について述べ，4章で，接尾辞配列により接尾辞トライの走査を模擬する方法について述べる．5章では，提案手法の有効性を実験により確認し，6章で関連研究を述べる．7章は結論である．

2. 準　　備

2.1 接尾辞等

アルファベット Σ に対して，Σ の要素を「文字」あるいは「最小単位」と呼ぶ．また，Σ の要素数（アルファベットの大きさ）を $|\Sigma|$ とする．

長さ n のテキスト T とは，文字列 $T = t_1 t_2 \dots t_n$ である．ここで $t_i \in \Sigma$ である．また，T 中の i 番目から j 番目 $(i \leq j)$ までの部分文字列は，$t_{i..j} = t_i t_{i+1} \dots t_j$ である．次に，T の位置 i から始まる接尾辞（suffix）とは，$t_{i..n}$ のことである．また，$T[i] = t_i$ である．

さらに T の部分文字列 $t_{i..j}$ からなる集合を $W(T) = \{X | X = t_{i..j}\}$ とする．$W(T)$ は，T のなかで異なる部分文字列全てからなる集合である．たとえば，$T = abab$ のとき，$W(T) = \{a, b, ab, ba, aba, bab, abab\}$ である．ここで，a, b，abなどは複数回 T 中に現れるが，$W(T)$ には一つずつしか含まれない．

2.2 接尾辞トライ

テキスト T の接尾辞トライとは，T の全ての接尾辞を保持しているトライのことである☆☆．このトライは，n 個の葉を持ち，それらは T 中の全接尾辞を表す．各葉は，それが表す接尾辞の開始位置をもち，T の全接尾辞が，左の葉から右の葉に Σ 上の辞書式順序で並んでいる．ここで，ある葉が接尾辞 $t_{i..n}$ を表すとすると，そのことは，根からその葉に至る長さ $n-i+2$ の道において，根からその葉までの各辺のラベルとして，$t_i, t_{i+1}, \dots, t_n, \$$ を付けることにより示される．ここで，$\$$は，$\$ \notin \Sigma$ であるような特別な区切記号である．**図1** には，“ABCABDABE” というテキストから作られる接尾辞トライを示す．

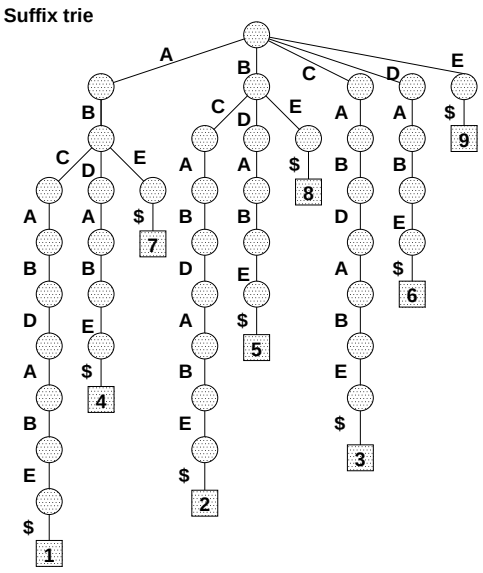


図1 テキストと接尾辞トライ．
Fig. 1 Text and its suffix trie.

なお，あるノードの深さとは，根からそのノードまでの道における辺の数である．また，先行順（preorder）の走査とは，1）根に訪問する，2）根に子供があれば左の子供から順番に先行順走査する，という順に再帰的にノードを訪問する方法を言い，後行順（postorder）の走査とは，1）根に子供があれば左の子供から順番に後行順走査する，2）根に訪問する，という順に再帰的にノードを訪問する方法を言う．

2.3 接尾辞配列

テキスト T の接尾辞配列 S とは，T の各接尾辞へのポインタを格納した一次元配列であり，各ポインタは，それが指し示す接尾辞の辞書式順序でソートされている．つまり，$S[i]$ は T の $S[i]$ 番目の接尾辞 $t_{S[i]..n}$ の開始位置であり，$i < j$ のときには，$t_{S[i]..n}$ の方が，

☆ 本稿では，接尾辞トライに基づく検索法を3章で概説するが，その方法は接尾辞木にも適用できる．

☆☆ 以下の用語,「トライ」「根」「葉」「ノード（節点）」「辺」「道」「子供（子）」の定義については文献17) を参照のこと．

	1	2	3	4	5	6	7	8	9
Text	A	B	C	A	B	D	A	B	E

Suffix array		*lcp* array		Suffixes denoted by $S[i]$
$S[1]$	1	$lcp[1]$	0	ABCABDABE
$S[2]$	4	$lcp[2]$	2	ABDABE
$S[3]$	7	$lcp[3]$	2	ABE
$S[4]$	2	$lcp[4]$	0	BCABDABE
$S[5]$	5	$lcp[5]$	1	BDABE
$S[6]$	8	$lcp[6]$	1	BE
$S[7]$	3	$lcp[7]$	0	CABDABE
$S[8]$	6	$lcp[8]$	0	DABE
$S[9]$	9	$lcp[9]$	0	E
		$lcp[10]$	0	

図2 テキストと接尾辞配列と *lcp* 配列.
Fig. 2 Text, its suffix array and *lcp* array.

$t_{S[j]..n}$ よりも，辞書順において，前に位置する．これは，接尾辞トライの葉のみを左から右に一次元配列に格納したものと等しい．

テキスト T の *lcp*（longest common prefix）配列とは，$n+1$ 個の整数からなる配列である．$lcp[i]$ は，$2 \leq i \leq n$ については，$t_{S[i-1]..n}$ と $t_{S[i]..n}$ との共通する先頭部分の（最長の）長さである．また，アルゴリズムの都合上，$lcp[1] = lcp[n+1] = 0$ と定義する．

図2には，テキスト “ABCABDABE” に対する接尾辞配列，および，それに対する *lcp* 配列を示す．この図で，たとえば，$lcp[2] = 2$ であるが，これは，$t_{S[1]..n} = ABCABDABE$ と $t_{S[2]..n} = ABDABE$ との共通先頭部分の長さである．図では，$lcp[i] > 0$ のときには，共通先頭部分に下線を引いてある．

なお，接尾辞配列 S において，$S[i]$ における深さ j の文字とは，$t_{S[i]+j-1}$ のことである．ただし，$S[i]+j-1 > n$ のときには “$” とする．たとえば，図2では，$S[4] = 2$ について，深さ1の文字は $t_{S[4]+1-1} = t_{S[4]} = t_2 = B$ であり，深さ2の文字は $t_3 = C$，深さ3の文字は $t_4 = A$ である．また，範囲 $[u..v]$ での深さ j といったときには，$u \leq i \leq v$ について，各 $S[i]$ における深さ j の文字からなる一次元配列を示すこととする．範囲を指定しないときには $u = 1, v = n$ である．たとえば，図2では，範囲 [3..7] の深さ1は “ABBBC” を示し，範囲 [3..7] の深さ2は “BCDEA” を示す．

2.4 空間領域

長さ n のテキスト T について，接尾辞配列と *lcp* 配列の占める空間領域は，テキストへの各ポインタを4バイトで表現するとすると，それぞれ，$4n$ バイトと $4(n+1)$ バイトである☆1．また，テキストの占める空間領域は，アルファベットの大きさにもよるが，$|\Sigma| \leq 256$ のときには n バイト，$|\Sigma| \leq 65536$ のときには $2n$ バイト，それ以上の大きさのアルファベットについては，$4n$ バイト必要である．よって，最大 $12n$ バイトにより，テキストと接尾辞配列と *lcp* 配列とを格納できる．

なお，接尾辞トライを構築するには，$O(n^2)$ のノードが必要となる．これは，もちろん，$12n$ バイトよりも多くの空間領域を必要とする．また，接尾辞トライのコンパクトな表現である接尾辞木を格納するには，$O(|\Sigma|n)$ だけの空間領域が必要である☆2．そのため，アルファベットとして日本語の文字集合のように大きな要素数を持つものを利用するときには，接尾辞木のサイズは非常に大きくなる [9),12),16)]．

3. 接尾辞トライを用いた近似文字列照合による全文検索の概要

本章では，文献13)に基づいて，接尾辞トライを用いた近似文字列照合による全文検索の概要を述べる．以下では，まず，近似文字列照合による全文検索を定義し，次に，動的計画法（dynamic programming）により二つの文字列間の編集距離（edit distance，略して距離とも言う）を求める方法☆3を示し，最後に，接尾辞トライを用いて近似文字列照合をする方法を示す．

3.1 近似文字列照合による全文検索

長さ m のパターンとは，文字列 $P = p_1 p_2 \ldots p_m$ のことである．近似文字列照合による全文検索とは，パターン P とテキスト T と許容値 $t(\geq 0)$ が与えられたとき，$W(T)$ の要素 X のなかで，パターン P との編集距離 $\mathrm{dist}(P, X)$ が t 以下であるような X 全てを見付けることであると定義する☆4．

3.2 動的計画法による編集距離の計算

文字列 $X = x_1 x_2 \ldots x_u$ と $Y = y_1 y_2 \ldots y_v$ の編集距離を $\mathrm{dist}(X, Y)$ とする．この距離は，X を Y に一致させる操作として，置換・削除・挿入を許し，それらのコストとして，x_i と y_j の置換コストを $\mathrm{sub}(x_i, y_j)$，

☆1 実際上は，*lcp* 配列の要素の最大値が65536を越えることはないため，*lcp* 配列は $2(n+1)$ バイトで格納可能である．
☆2 なお，これは，それぞれのノードの子ノードをアルファベットサイズの配列として作成する場合であり，2分木などで表した場合には $O(n)$ の空間領域が必要である．
☆3 動的計画法を用いた距離計算については，詳しくは，たとえば，文献6)を参照のこと．
☆4 この定義では，X の T 中での出現位置は検索しない．しかし，本稿で述べるアルゴリズムを修正して出現位置も検索するようにすることは容易である．

x_iの削除コストを$\mathrm{del}(x_i)$，y_jの挿入コストを$\mathrm{ins}(y_j)$としたとき，XをYに一致させるために必要な操作系列のコストの和の最小値として定義される．

$\mathrm{dist}(X,Y)$を計算するために，$x_{1..i}$と$y_{1..j}$との距離を$D[i,j]$と定義する．このとき，$\mathrm{dist}(X,Y)=D[u,v]$である．$D[i,j]$は以下のように定義できる．

$$
\begin{aligned}
&D[0,0]=0,\\
&D[i,0]=D[i-1,0]+\mathrm{del}(x_i),\\
&D[0,j]=D[0,j-1]+\mathrm{ins}(y_j),\\
&D[i,j]=\min\left(\begin{array}{l}D[i-1,j-1]+\mathrm{sub}(x_i,y_j)\\ D[i-1,j]+\mathrm{del}(x_i)\\ D[i,j-1]+\mathrm{ins}(y_j)\end{array}\right).
\end{aligned}
$$

ただし，$1\le i\le u$，$1\le j\le v$である．また，$\mathrm{sub}(x_i,y_j),\mathrm{del}(x_i),\mathrm{ins}(y_j)\ge 0$であり，かつ，$x_i=y_j$のときには，$\mathrm{sub}(x_i,y_j)=0$である．なお，$D$のことを距離行列と呼ぶ．

これらの式の解釈は以下の通りである．まず，$D[0,0]$は，空文字列と空文字列との距離である．次に，$D[i,0]$は，$\sum_{1\le k\le i}\mathrm{del}(x_k)$と等価であるが，これは，$x_{1..i}$と（$y_1$の前の）空文字列とを一致させるには，$x_{1..i}$を削除する必要があることを示す．また，$D[0,j]$は，$\sum_{1\le l\le j}\mathrm{ins}(y_l)$と等価であるが，これは，（$x_1$の前の）空文字列と$y_{1..j}$とを一致させるには，$y_{1..j}$を$x_1$の前に挿入する必要があることを示す．

次に，$D[i,j]$を計算するには，まず，$x_{1..i-1}$と$y_{1..j-1}$等の距離計算が済んでいるとする．この先，距離計算を続けていくとして，x_iとy_jについて可能な操作は，x_iとy_jを置換するか，x_iを削除するか，y_jを（$x_{1..i}$の後に）挿入するかしかない．従って，距離$D[i,j]$を求めるには，これらの操作の中で，これまでに計算済みの距離とコストの和が最小となる操作を選択すれば良い．

なお，以下では，挿入／削除コストを1，置換コストを同一文字には0異なる文字には1としたときに計算される編集距離を「コスト1の距離」と呼ぶ．

3.3 距離計算の打ち切り

二つの文字列間の距離を計算するとき，ある許容値tを定めて，それよりも距離が大きくなったら距離計算を打ち切ることにすると，距離計算の手間を省くことができる．これは以下のアルゴリズムによる[14]．

距離行列を計算するときに，列jを0から順に大きくしながら，$D[i,j]$を計算するが，その計算のとき，$D[i,j]\le t$を満たすような最大のiを$C[j]$として記録しておく．もし，全てのiについて，$D[i,j]>t$なら，$C[j]=0$とする．次に，$j+1$列目の値を計算するときには，0行から$C[j]+1$行目までについて，距離を計算すれば十分である．なぜなら，$C[j]+2$行目以降については，距離がtより大きいことが分かっているので，デフォルト値として，$t+1$を与えれば十分であるからである[14]．なお，$C[0]$は，$D[i,0]\le t$を満す最大のiである．

		0	1	2	3	4	5	6	7
		ϕ	a	c	d	f	b	d	f
0	ϕ	0	1	2	3	4	5		
1	a	1	0	1	2	3	4		
2	d	2	1	1	1	2	3		
3	f			2	2	1	2		
4	d						2		

図3 打ち切りを伴なう距離計算．文献13) Table 2 の一部を若干変更したもの．
Fig. 3 Distance calculation with cut-off. Modification of a part of Table 2 in 13).

打ち切りを伴なう距離計算の例として，$X=adfd$と$Y=acdfbdf$に対して，コスト1の距離について，許容値を1として，このアルゴリズムにより距離行列を計算する過程を次の段落に示し，その結果を**図3**に示す．なお，図3の空白部分には，デフォルト値（=許容値+1=2）が入っているとする．

まず，$j=0$について，$D[0,0]=0,D[1,0]=1,D[2,0]=2$である．ここで，$D[2,0]>1$なので，計算を打ち切り，$C[0]=1$とする．$j=1$については，$C[0]+1=2$までの値，$D[0,1]=1,D[1,1]=0,D[2,1]=1$を計算し，$C[1]=2$とする．同様に，$C[2]=2$, $C[3]=2,C[4]=3$となる．$j=5$については，全ての距離が1より大きいので，$C[5]=0$となる．この時点で，$D[4,5]>1$であるので，$D[4,7]>1$であることが分かる．したがって，この時点で全体の距離計算を打ち切ることができる．

3.4 接尾辞トライを用いた近似文字列照合

3.4.1 接尾辞トライの性質

接尾辞トライの次の二つの性質[13]を利用すると，パターンPとテキストTと許容値tが与えられたとき，PとTの全ての接尾辞との距離計算を効率的に実行できる．

性　質　1

トライにおけるあるノードをvとすると，根からvへの道は，根からvを通って葉に至るまでの道に対応する全ての接尾辞に共有されている．つまり，根からvまでの道は，それら接尾辞の共通の先頭部分と対応している．そのため，それら接尾辞とパターンとの

距離行列については，根からvに至るまでの先頭部分においては，等しい距離が格納された列が得られる．たとえば，図1においては，たとえば，接尾辞 “ABCABDABE” と “ABDABE” と “ABE” とは，先頭部分の “AB” を共有しているので，これらについての距離行列は，“AB” に関する列を共有する．

そのため，トライを根から先行順に辿り，その際に，大域的に保持される一つの距離行列を一列ずつ更新していけば，あるノードに到達したときには，そのノードに関連する全ての接尾辞について，根からそのノードまでの共有する先頭部分について（パターンとの）距離が計算されていることになる．

性　質　2

ある接尾辞の距離行列について，もし，距離行列のj列において，全ての距離が許容値tよりも大きいとしたら，$j+1$列目以降まで列を伸ばしていったとしても，やはり，距離は，tよりも大きい．そのため，一旦，距離がtよりも大きくなったら，それ以上，トライを辿る必要はないので探索を打ち切ることができる．

3.4.2 検索アルゴリズム

これら二つの性質を利用した検索アルゴリズムをC風の擬似コードとして**図4**に示す☆．図4では，main において，距離行列の計算のための初期化をし，その後で，根の全ての子供に対して再帰的に search を呼出す．search を再帰的に呼出すことにより，トライを先行順に辿ることができる．search は，先行順にトライを巡回する際に，距離行列の値を計算し，その値に応じて不要な枝を刈り込む．

search(node,j) においては，

(1) まず，node が葉かどうかを調べ，もし葉であるなら，既にパターンと接尾辞との比較は終了しているので，探索を打ち切る．
(2) 次に，cutp(T(node),j)は，距離行列のj列目の値を計算し，打ち切り可能なときには真（true），そうでないときには偽（false）を返す．
(3) もし，パターンと，根からnodeまでの道に対応する文字列との距離D[m,j]が許容値t以下なら，パターンの検索に成功したことになる．
(4) 最後に，nodeの全ての子供に対して，再帰的にsearchを適用する．

上述の（2）におけるcutp(T(node),j)での処理は以下のようなものである．

```
/* 変数などの説明 */
Dは3.2節における距離行列
Cは3.3節において距離計算の打ち切りに利用した配列
Pはパターン．P[i]はパターンのi番目の文字(1<=i<=m)
T(node)はnodeとその親ノードを結ぶ辺に対応する文字
tは許容値
/* アルゴリズム */
main(){
  /* 初期化 */
  C[0] := D[0,0] := 0
  for(i:=1; i<=m; i++){
    D[i,0] := D[i-1,0] + del(P[i])
    if(D[i,0]<=t){C[0]:=i}else{break}
  }
  /* 検索 */
  foreach child in 根の全子供 {search(child, 1)}
}
search(node, j){
  if (nodeが葉) return
  if (cutp(T(node), j)) return
  if (D[m,j]<=t) 検索成功
  foreach child in (nodeの全子供) {
    search(child, j+1)
  }
}
cutp(textChar, j){
  /* パターンに対するデフォルト距離 */
  D[m,j] := t+1
  /* 距離行列におけるj列目の距離を計算する */
  C[j] := 0
  D[0,j] := D[0,j-1] + ins(textChar)
  for(i:=1; i<=min(C[j-1]+1, m); i++){
    D[i,j] := min(D[i-1,j-1]+sub(P[i],textChar),
                  D[i-1,j]+del(P[i]),
                  D[i,j-1]+ins(textChar))
    if(D[i,j]<=t){C[j] := i}
  }
  /* 打ち切り可能性を判定する */
  if(C[j]=0){
    return true
  }else{
    /* j+1列目の計算に必要なデフォルト値 */
    if(C[j]>C[j-1]){D[C[j]+1,j] := t+1}
    return false
  }
}
```

図4　近似文字列照合による全文検索のアルゴリズム．
Fig. 4　Full-text approximate string matching algorithm.

(1) D[m,j]:=t+1としてデフォルト値を設定する．この値を設定する理由は，3.3節で説明したように，打ち切りを伴なう距離計算の場合には，必ずしもD[m,j]が計算されないからである．
(2) 次に，距離行列におけるj列目の距離を計算する．このとき，同時に，C[j]を計算する．
(3) もし，C[j]=0なら，全ての距離が許容値より大きく，距離計算の打ち切りが可能なので，trueを返す．そうでないなら，j+1列目の計算に必要なデフォルト値を設定してからfalseを返す．

j+1列目の計算に必要なデフォルト値については，**図5**に示す距離行列を利用して説明する．まず，C[j]>C[j-1]，つまり，C[j]=C[j-1]+1の場合を考える☆☆．一般に，D[i,j+1]の計算には，D[i-1,j]，D[i-1,j+1]，D[i,j]が必要である．j列までの計算に

☆ ただし，代入演算子を “:=” とし，比較演算子の等号を “=” としている．

☆☆ C[j]の取り得る値は，0以上 C[j-1]+1以下の整数値である．

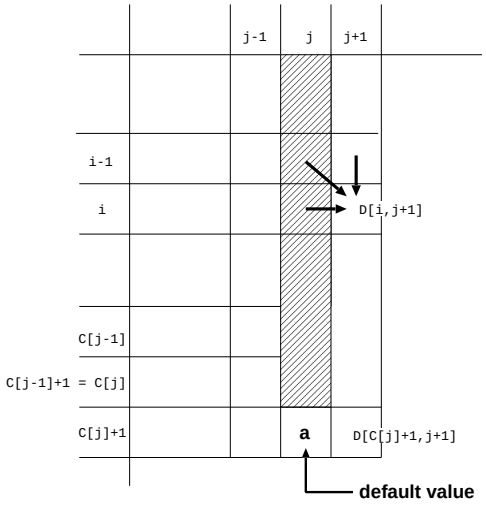


図5 距離のデフォルト値の取り扱い.
Fig. 5 Handling default values for distances.

		0	1	2
		ϕ	A	B
0	ϕ	0	1	2
1	D	1	1	2
2	C	2	2	2
3	A			

図6 “DCA” と “AB” との距離行列（探索が打ち切られる場合）.
Fig. 6 Distance matrix between “DCA” and “AB” (cut-off).

		0	1	2	3
		ϕ	B	C	A
0	ϕ	0	1	2	3
1	D	1	1	2	3
2	C	2	2	1	2
3	A				1

図7 “DCA” と “BCA” との距離行列（検索に成功した場合）.
Fig. 7 Distance matrix between “DCA” and “BCA” (success in approximate string matching).

よると，図5の斜線部の値（`D[0,j]`,..., `D[C[j],j]`）は`cutp`での`for`ループの終了時点で計算済みである．そのため，`j+1`列目において，`0<=i<=C[j]`については，`D[i,j+1]`を問題なく計算できるが，`D[C[j]+1,j+1]`を計算するためには，`for`ループで計算されていない図5の`a`の値が必要である．ところで，この値は，3.3節でも述べたように，許容値`t`よりも大きいことが分かっているので，デフォルト値として，`t+1`を`D[C[j]+1,j]`に代入すれば良い．なお，`C[j]<=C[j-1]`のときには，`j+1`列目の計算は，デフォルト値を使わなくても可能である．

接尾辞トライを用いた近似文字列照合の例として，テキスト “ABCABDABE” から作られる接尾辞トライ（図1）に対して，パターン “DCA” を検索した場合を考える．このとき，根から “AB” までを辿ったときの，コスト1の距離による距離行列は，**図6**のようになる．そのため，距離の許容値が1の場合には，この時点でトライの探索を打ち切ることができる．次に，たとえば，トライを “BCA” まで辿ったときの距離行列を**図7**に示す．このときには，`D[3,3]=1`であり，この値は許容値以下であるので，“BCA” はパターンに近似的に一致したとして検索される．なお，図6と図7において，空欄となっている部分は，距離のデフォルト値（許容値+1=2）が入っているものとする．

4. 接尾辞配列による接尾辞トライの先行順走査の模擬

本章では，まず，接尾辞トライにおける先行順走査を，接尾辞配列で模擬する二つの方法を述べる．それらのうちで最初に述べる方法は，従来研究[1),2),12)]で用いられている，2分探索を利用する方法であり，次に述べる方法が，本稿で提案する，*lcp*配列を用いて先行順走査をする方法である．それらの方法を述べた後で，提案手法による先行順走査を用いた近似文字列照合による全文検索アルゴリズムを示す．

4.1 2分探索による接尾辞配列の先行順走査

接尾辞トライの先行順走査を接尾辞配列により模擬する方法としては，接尾辞配列を2分探索することにより，接尾辞トライにおけるノードと対応する接尾辞配列の範囲を見付け，その範囲に基づいて先行順走査を模擬するというものが考えられる．これは，従来研究[1),2),12)]で用いられている方法である．

接尾辞配列の各要素と接尾辞トライの葉とは一対一に対応するので，接尾辞配列は接尾辞トライを表現できる．たとえば，図1の接尾辞トライと図2の接尾辞配列とは一対一に対応している．

また，接尾辞配列は，接尾辞が辞書順にソートされているので，2分探索することによりトライの走査を模擬できる．たとえば，図2において，“BCA” を，厳密一致（exact match）で検索したいとする．そのときには，まず，深さ1で，“B” を2分探索すると，それが，[4..6] の範囲にあることがわかるので，次に，その範囲で，深さ2において，“C” を検索すると，それが [4..4] の範囲にあることがわかり，最後に，深さ3において，“A” を検索すると，これも [4..4] の範囲にあ

るので，“BCA” が図2の接尾辞配列にあることが分かる．これは，図1の接尾辞トライにおいて，根から，検索文字列が “BCA” となるようにノードを辿っていくこと同等である．このように，接尾辞配列の範囲と深さを合わせたものは，接尾辞トライの辺に一対一対応する（したがって，各辺の子供のノードとも一対一対応する）ことが分かる．

上記は，厳密一致における接尾辞配列の探索例であるが，近似文字列照合をするには，接尾辞トライでノードの子供全てを枚挙していたのと同様なことを，接尾辞配列でもしなければならない．たとえば，図1の接尾辞トライの根に相当する，図2の接尾辞配列の全範囲においては，根の子供（につながる辺）として，深さ1の範囲 [1..3] に “A”, 範囲 [4..6] に “B”，範囲 [7..7] に “C”, 範囲 [8..8] に “D”, 範囲 [9..9] に “E” があることが分かる必要がある．もしそのような範囲が分かれば，その各々の範囲に対して，再帰的に探索をすることにより，接尾辞トライの先行順の走査を模擬できるので，近似文字列照合ができる．

このような範囲を枚挙することは，接尾辞配列の左端（先頭）の方から順に2分探索することにより可能である☆．まず，上記の例では，$S[1]$ における深さ1の文字が “A” であるので，“A” の右端を2分探索すると，“A” は深さ1で範囲 [1..3] にある．次に，$S[4]$ における深さ1の文字は “B” であるので，“B” の右端を2分探索すると，“B” は深さ1で範囲 [4..6] にある．同様にして，“C” の範囲は [7..7]，“D” の範囲は [8..8]，“E” の範囲は [9..9] である．このようにして範囲を求めたら，後は，各範囲について，再帰的に，同様な手続きをすることにより，接尾辞配列により，接尾辞トライの先行順の走査を模擬できる．そのため，図4の検索アルゴリズムを利用して，接尾辞配列により近似文字列照合をすることができる．

4.2 *lcp* 配列を用いた接尾辞配列の先行順走査

本節では，*lcp* 配列を用いて，接尾辞配列を先行順に走査する方法を提案し，次節で，それを利用して近似文字列照合をするアルゴリズムを示す．

まず，図1と図2とを比較することにより，$lcp[i]$ が，接尾辞トライにおいて，$S[i-1]$ に対応する葉と $S[i]$ に対応する葉とについて，それぞれ，根から葉までの道を考えたとき，それら二つの道の共有部分の長さ（辺の数）であることが分かる．たとえば，図2において，$t_{S[1]\ldots n} = t_{1\ldots n} = ABCABDABE$ と $t_{S[2]\ldots n} = t_{4\ldots n} = ABDABE$ とは，“AB” を共有しているが，このことは，図1においては，それら接尾辞に対応する道において，先頭の “AB” の部分が共有されていることに対応する．そして，このときの共有部分の長さは $lcp[2] = 2$ である．

このことを利用すると，次のようなコード断片（これをコード1と呼ぶ）により，接尾辞トライの全ての辺に対応する文字を，先行順走査に従って印刷できる．

```
for(i:=1; i<=n; i++){
  for(k:=lcp[i]; S[i]+k<=n; k++){
    printf("%c", T[S[i]+k]);
  }
}
```

次に，コード1に，S や *lcp* などの値も出力させるようにして，かつ，コード1では出力されない文字を “*” として出力させるようにすると，下記コードとなる．

```
for(i:=1; i<=n; i++){
  printf("S[%d]=%d, lcp[%d]=%d: ",
         i, S[i],i,lcp[i]);
  for(k:=0; k<lcp[i]; k++){printf("* ");}
  for(k:=lcp[i]; S[i]+k<=n; k++){
    printf("%c ", T[S[i]+k]);
  }
  printf("\n");
}
```

これを，図2に適用すると以下の結果が得られる．

```
S[1]=1, lcp[1]=0: A B C A B D A B E
S[2]=4, lcp[2]=2: * * D A B E
S[3]=7, lcp[3]=2: * * E
S[4]=2, lcp[4]=0: B C A B D A B E
S[5]=5, lcp[5]=1: * D A B E
S[6]=8, lcp[6]=1: * E
S[7]=3, lcp[7]=0: C A B D A B E
S[8]=6, lcp[8]=0: D A B E
S[9]=9, lcp[9]=0: E
```

この結果では，図2において下線となっている部分が “*” となっている．つまり，図2において下線となっている部分は，コード1では印刷されない部分である．つまり，その部分は，コード1では評価されていない．そして，その部分は，接尾辞トライにおいては，既に先行順で通っている部分であるので，評価されないのが適切である．

コード1を，図2の場合にトレースすると以下のようになる．まず，`i=1` のときには，`lcp[1]=0` であるので，`T[S[1]+0]=T[1]` から `T[n]=T[9]` まで，すなわち，“ABCABDABE” が内側の `for` ループで印刷される．次に，`i=2` のときには，`lcp[2]=2` であるので，`T[S[2]+2]=T[6]` から `T[n]=T[9]` まで，すなわち，“DABE” が印刷される．また，`i=3` のときには，

☆ このような範囲を枚挙する方法の他のものとしては，まず，接尾辞配列の中央部分を範囲内に含むような子供（に対応する範囲）を2分探索により探し，その後で，接尾辞配列を，その子供の前の範囲，その子供の範囲，その子供の後の範囲，というように3分割し，次に，その子供の前と後の範囲に対して，それぞれ，再帰的に同様な探索をすることにより子供（に対応する範囲）全てを枚挙するというものがある [12]．しかし，その方法によっても計算量は同じである．本稿では，左端から探索していく方法の方が実装が容易なため，左端から探索する方法を採用した．

```
main()
{
  /* 初期化 */
  (図4の対応部分と同じ)
  /* 検索 */
  for(i:=1; i<=n; i++){
    cut := false
    for(k:=lcp[i]; S[i]+k<=n; k++){
      j := k + 1
      if(cutp(T[S[i]+k], j)){
        cut := true
        break
      }
      if (D[m,j]<=t) 検索成功
    }
    if(cut){ /* 接尾辞のスキップ */
      while(lcp[i+1]>=j){i++}
    }
  }
}
```

図8 *lcp*配列を用いた近似文字列照合による全文検索のアルゴリズム.
Fig. 8 Full-text approximate string matching algorithm using a *lcp* array.

lcp[3]=2なので，T[S[3]+2]=T[9]であるので，“E”のみが印刷される．以下，同様にして，コード1は，接尾辞トライを先行順に走査したのと同じ順番で，接尾辞トライの各辺に対応付けられた文字を印刷する．

4.3 *lcp*配列を用いた近似文字列照合による全文検索のアルゴリズム．

*lcp*配列を用いた検索アルゴリズムは，基本的には，コード1における印刷の部分を，近似文字列照合のための処理に置き換えたものである．**図8**には，そのアルゴリズムを示す．図8において明示されていない部分（cutpなど）は，図4のアルゴリズムと同じとする．

図8における2重のforループの外側のループでは，接尾辞配列Sの先頭から順番に全ての接尾辞についてアクセスする．内側のループでは，kを，(lcp[i]未満については距離行列の行の値は計算済みなので)，lcp[i]から順番に増やしながら，S[i]の深さk+1の文字T[S[i]+k]について，距離行列のk+1列目（j列目）を計算し，距離計算の打ち切りが可能ならば，ループを抜け出す．距離計算が打ち切られたときには，距離計算が打ち切られた接尾辞と先頭部分を共有する接尾辞をwhileループによりスキップする．whileループにおいて，jは，距離計算が打ち切られた接尾辞の先頭部分の長さである．一方，lcp[i+1]は，現在距離計算が打ち切られた接尾辞と，それに（接尾辞配列上で）後続する接尾辞との先頭部分の共通部分の長さである．そのため，もし，lcp[i+1]>=jであるなら，後続する接尾辞は，現在距離計算が打ち切られた接尾辞の先頭部分を，その先頭部分に持つ．そのため，そのような接尾辞について距離計算はしなくても良いためスキップする．なお，スキップされる接尾辞の数が非常に大きい場合には，(whileループによる）線形探索によりスキップすることは不利になる可能性がある☆．しかし，5章の実験で示すように，日本語の文字や英語の単語をアルファベットとするテキストにおいては，whileループによりスキップされる数は小さいため，線形探索によりスキップしても十分である．

3.4.2節での例と同じく，コスト1の距離について許容値を1として，図2のテキスト“ABCABDABE”について，パターン“DCA”を検索した場合について，図8のアルゴリズムの実行過程の概要を以下に示す．

まず，i=1の場合には，内側のforループでは，パターンと接尾辞“ABCABDABE”を比較していく．このときには，lcp[1]=0であるので，kは0を初期値とする．また，T[S[1]+0]=T[1]は“A”であるので，まず，cutp("A",1)を計算する．この場合，距離行列では，図6の1列目が得られる．次に，k=1のときには，cutp("B",2)が計算される．そして，距離行列では，図6の2列目が得られる．この場合には，cutpはtrueを返すので，内側のforループを抜け出す．この時点で，“AB”を先頭部分に持つ接尾辞は，パターンとは一致しないことが分かるので，そのような接尾辞は，whileループにおいてスキップする．この場合には，j=2であり，かつ，lcp[2]=lcp[3]=2であるので，i=2,3の場合について，それぞれの接尾辞“ABDABE”と“ABE”とが，先頭部分“AB”を共有するので，スキップする．

そのため，次に内側のforループを実行するときには，i=4である．これについては，パターンと接尾辞“BCABDABE”とを比較していくと，“BCA”について検索が成功する（図7）が，次に，“BCAB”において，距離計算の打ち切りが起きる．

次に，i=5について，内側のforループを実行する．この場合には，lcp[5]=1であるので，kは1から始まる．そのため，既にi=4のときに計算済みであるcutp("B",1)については計算をすることなく，cutp("D",2)から計算することができる．

以下，同様にして実行していくと，コスト1の距離について許容値を1として，テキスト“ABCABDABE”について，パターン“DCA”を検索した場合には，“BCA”，“CA”，“DA”が距離1で検索される．

4.4 計 算 量

パターンの長さをm，テキストの長さを$n(\gg m)$，許容値をtとしたとき，領域計算量，時間計算量，空間領域（索引の大きさ）は以下のようになる．

☆ その場合には，線形探索以外の方法，例えば文献9)の方法でスキップする接尾辞の右端を探索した方が良い．

4.4.1 領域計算量

4.1節で述べた従来手法，および，4.3節で述べた提案手法ともに，支配的な領域を占めるのは距離行列である．図4における`cutp`をみると，$1 \leq i \leq m$であり，$1 \leq j \leq m+t+1$である．ここで，$j \leq m+t+1$である理由は，$j = m+t+1$のときには，パターンとの距離が必ずtを越えるからである．そのため，距離行列の占める領域は$O(m(m+t))$である．このオーダは，従来手法と提案手法で同じである．

4.4.2 時間計算量

提案手法では，図8において，$1 \leq i \leq n$であり，$0 \leq k \leq m+t$である．また，`cutp`では，$1 \leq i \leq m$である．よって，$O(n(m+t)m)$が最悪時の時間計算量となる．一方，従来手法では，トライを辿る深さの最大値が$m+t+1$であるので，訪問されるトライのノード数のオーダは$O(|\Sigma|^{m+t})$である．また，各訪問されたノードで，`cutp`を呼ぶので，その時間計算量が$O(m)$である．更に，子供のノードを2分探索するのに$O(\log n)$かかる．よって，$O(m|\Sigma|^{m+t}\log n)$が最悪時の時間計算量となる．

平均時の時間計算量は以下のようである．まず，探索されるトライのノードの平均の深さは$O(t)$であり，`cutp`の平均的な時間計算量は$O(t)$であるので，$O(t|\Sigma|^t)$が接尾辞トライを用いた場合の時間計算量である[13]．したがって，従来手法の時間計算量は$O(t|\Sigma|^t \log n)$である．一方，提案手法においては，訪問されるノードの数は従来手法と同じだが，ノードの探索（図8の`while`ループでの接尾辞のスキップ）が生じるのは，距離計算の打ち切りが発生した場合のみである．打ち切りが起きるノード数のオーダは$O(|\Sigma|^t)$であるので，スキップされる接尾辞数の平均をsとすると，スキップにより生じる計算量は$O(s|\Sigma|^t)$である．この計算量が加算された$O((t+s)|\Sigma|^t)$が全体の計算量となる．ここで，sについては，もし線形探索でなく，文献9)の探索方法を使うと，$O(\log n + t)$となるので，全体の計算量は$O((t+\log n)|\Sigma|^t)$である．

従来手法と提案手法の平均時の時間計算量を比べると，従来手法においては，ノードの探索のための計算量が乗算されるが，提案手法では，それが加算される．そのため，提案手法の方が計算量が少なくて済む．

4.4.3 空間領域

提案手法の空間領域は2.4節で述べたように$12n$バイトである．一方，従来手法では，*lcp*配列を使わないので，$8n$バイトである．そのため，空間領域は検索速度とトレードオフの関係にあると言える．

ただし，*lcp*配列は，近似文字列照合だけに役立つのではなく，その他にも，単語列の頻度などの統計量を得るのに役立つ[7),10),16)]．また，*lcp*配列を利用することにより，厳密な一致による検索も高速にできる[9)]．つまり，近似文字列照合という文脈以外からも*lcp*配列の有効性は確認されている．そのため，*lcp*配列を索引として持つ意義はあると考える．

5. 実　　験

本章では，4章で述べた二つの走査法を用いた場合について，検索速度を比較する．

5.1 実験材料

5.1.1 コーパス

検索対象のコーパス☆は，日本語と英語のコーパスが一つずつである．日本語のコーパスとしては，CD毎日新聞98,99年度版を用い，英語のコーパスとしては，BNC（British National Corpus）を用いた☆☆．これらのコーパスからは，生のテキストに相当する部分のみを抽出し，記事番号などのメタ情報は除去した．

日本語コーパスの最小単位は文字とし，英語コーパスの最小単位は単語とした．このような単位を設定した理由は，我々が近似文字列照合による全文検索をするときの主要な目的が類似用例の検索であり，そのためには，これらの単位が有効である[3),4)]からである．

それぞれのコーパスは，まず，1番目から6番目までの部分コーパスに分割した．その分割の方法は，コーパスのi番目の行が，iを6で割ったときの余り$+1$番目の部分コーパスに含まれるようにするというものである．そのあと，検索対象のコーパスとして，部分コーパス1からなるもの，部分コーパス1と2からなるもの，...，部分コーパス1,2,3,4,5からなるもの，という五つのコーパスを作った．これらのコーパスをSize-1,Size-2,...,Size-5と呼ぶことにする．なお，部分コーパス6は，パターンの抽出に利用した．

表1には，毎日新聞とBNCについて，Size-1からSize-5について，異なり単位数と延べ単位数を示す．なお，14.9Mなどの Mは，10の6乗のことである．

☆ これまでの説明では「テキスト」という言葉により検索対象を表現したが，本章では「コーパス」という言葉により検索対象を表現する．コーパスは，文字からなるという点で，テキストの一種であるので，そのまま本章でも「テキスト」という言葉を使っても良いのだが，毎日新聞やBritish National Corpusなどは，著者らが主な研究領域としている自然言語処理においては「コーパス」と呼ばれるため「コーパス」という言葉を使う．

☆☆ BNCには，新聞や小説などの書き言葉からなるテキストが90%あり，残りの10%は話し言葉を書き起こしたテキストである．本実験では，BNCの書き言葉の部分のみを用いている．

表1 異なり単位数と延べ単位数.
Table 1 Number of different units (types) and the number of total units (tokens).

		Size-1	Size-2	Size-3	Size-4	Size-5
毎日新聞	異なり文字数	4,296	4,608	4,783	4,880	4,965
	延べ文字数	14.9 M	29.8M	44.8M	59.7M	74.6M
BNC	異なり単語数	251,491	362,443	449,288	523,408	588,082
	延べ単語数	16.7M	33.5M	50.2M	66.9M	83.6M

5.1.2 パターンと距離関数

パターンは部分コーパス6からランダムサンプリングした．このとき，毎日新聞については，6,12,18の長さの文字列を，それぞれ100ずつ抽出し，BNCについては，3,6,9の長さの単語列を，それぞれ100ずつ抽出した．なお，これらの文字列や単語列としては，句読点等を含まないものを選んだ．

距離としては，コスト1の距離を利用した．また，許容値としては，毎日新聞については，長さ6の文字列については2，長さ12の文字列については2と4，長さ18の文字列については2,4,6を試した．BNCについては，長さ3の単語列については1，長さ6の単語列については1と2，長さ9の単語列については1,2,3を試した．

5.1.3 プログラム

検索用のプログラムはC言語で記述した．4.1節で説明した2分探索を利用する手法（これを *binsearch* 法と呼ぶ）と4.3節で説明した提案手法（これを *lcp* 法と呼ぶ）とを実装するにあたり，共有できるコードは共有し，二つの実装の違いは，接尾辞配列の走査法の違いだけになるように努めた．また，図4や図8のアルゴリズムにおける「検索成功」の時点では，通常は，検索された文字列を印刷するなどの処理をするのだが，本実験では，検索時間を測定するのが目的であるので，この部分では何の処理もしていない．

5.1.4 計算機環境

実験に用いた計算機環境は以下の通りである．

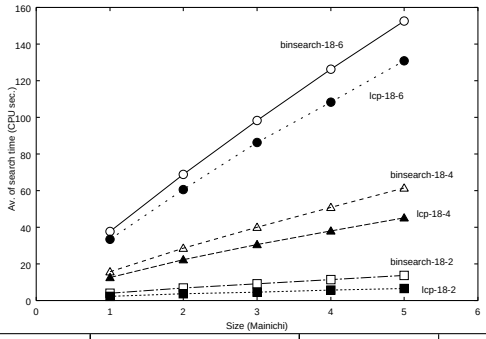
CPU Alpha 21264（クロック速度 750MHz） x 2
メモリ 2048MB
OS Linux 2.2.14 （Kondara MNU/Linux release 1.1）

なお，本実験での検索時間はCPU時間☆により計測されている．

☆ 一般に，CPU時間は，ユーザプロセスにより消費された時間とシステム（OS）により消費された時間の和，つまり，

CPU時間 = ユーザCPU時間 + システムCPU時間

のようになっているが，本実験で計測したCPU時間は，ユーザCPU時間のみである．つまり，本実験では，ユーザCPU時間のことをCPU時間と言っている．



許容値	2			4		6
長さ	6	12	18	12	18	18
binsearch	13.7	13.8	13.7	61.3	61.3	153
lcp	6.6	6.5	6.7	45.3	45.3	131

図9 毎日新聞での平均検索CPU時間（秒）（Size-1〜5のグラフとSize-5における表）.
Fig. 9 Average search CPU time (seconds) on Mainichi (graphs of Size-1 to 5 and the table of Size-5).

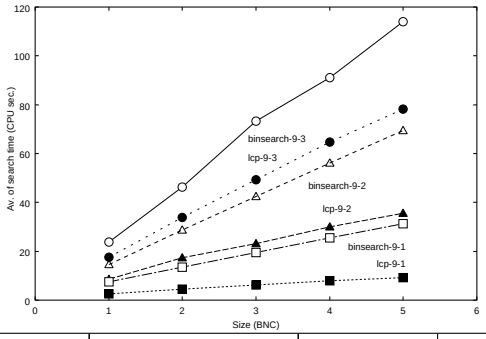
5.2 実験結果

まず，平均検索時間を比較する．次に，*lcp* 法において探索打ち切り時にスキップされる接尾辞の数を調べる．

5.2.1 平均検索時間

図9と**図10**には，毎日新聞とBNCについての1パターンあたりの平均検索CPU時間（秒）を示す．これらの図では，各線のパターンの性質を，“検索方法-パターンの長さ-許容値” のように示している．たとえば，binsearch-18-6は，*binsearch* 法により，パターンの長さが18，許容値が6で検索した場合の平均検索時間である．また，これらの図では，Size-5における検索時間を表に示してある．これらの表において，*binsearch* や *lcp* の行にある数字は，それぞれ，*binsearch* 法や *lcp* 法における平均検索時間である．

これらの図では，最長のパターン（図9では長さ18，図10では長さ9）についてしか平均検索時間のグラフを示していないが，その理由は，平均検索時間が，パターンの長さに関らず，ほとんど変化しないからである．ただし，「ほとんど変化しない」とは，それぞれのコーパスにおいて，検索法と許容値が同じで長さの



許容値	1			2		3
長さ	3	6	9	6	9	9
binsearch	31.3	31.3	31.3	68.9	69.4	114
lcp	9.2	9.1	9.2	35.3	35.6	78.3

図10 BNCでの平均検索CPU時間（秒）（Size-1〜5のグラフとSize-5における表）.
Fig. 10 Average search CPU time (seconds) on BNC (graphs of Size-1 to 5 and the table of Size-5).

みが違うパターンの組（たとえば，binsearch-18-6 とbinsearch-12-6）について，平均検索時間をt_1,t_2とすると，

$$\text{検索時間の差の全体に占めるパーセンテージ} = 100 \times \frac{|t_1 - t_2|}{(t_1 + t_2) * 0.5}$$

としたとき，このようなパターンの組全てに対して，検索時間の差の全体に占めるパーセンテージの平均を求めると，それが，毎日新聞については0.49%になり，BNCについては1.08%になるということである．

このように，検索時間がパターンの長さに依存しないということは，接尾辞トライにおいて訪問されるノードの数がパターンの長さに依存しないということを示している（なぜなら検索時間は訪問したノード数に比例するから）．実際，3.3節で述べた距離計算の打ち切り法を利用した場合には，許容値tに対して，探索されるトライのノードの平均の深さは，$O(t)$であり，パターンの長さには依存しない[13]．そのため，訪問されるノードの数，したがって，検索時間もtのみに依存し，パターンの長さには依存しない．

また，探索されるノードの平均の深さが$O(t)$であることから，検索時間はtに指数的に増加すると言える．なぜなら，訪問されるノードの数は，一般に，ノードの深さに指数的に増加するからである．このことは，図9と図10の中にあるそれぞれの表からも分かる．

最後に，*lcp*法と*binsearch*法の検索速度を比べると，*lcp*法の方が速いこと（*lcp*法の方が検索時間が短かいこと）が図9と図10とから分かる．特に，許容値が小さいときに，その差は，顕著である．たとえば，図9では，許容値が2のときには，*lcp*法は*binsearch*法の約2倍の検索速度（1/2の検索時間）である．なお，許容値が大きくなると，両者の検索速度の比は小さくなるが，検索時間の差は無視できない程の大きさである．たとえば，図9の表では，*binsearch*法と*lcp*法の検索時間の差は，許容値2,4,6に対して，7, 16, 22秒であり，図10の表では，許容値1,2,3に対して，検索時間の差は，22, 34, 36秒となっている．また，図9と図10のグラフから，コーパスの大きさが大きくなっていくと，検索時間の差も大きくなっていくことが分かる．これらより，*lcp*法は，検索速度の向上に役立つと言える．

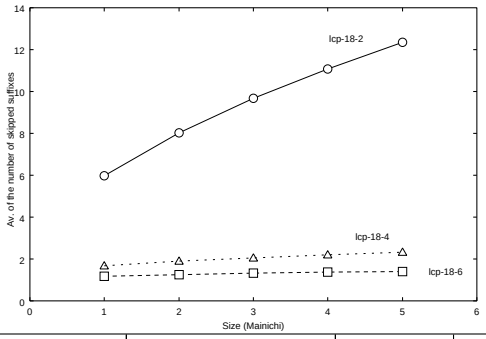
これらの検索速度差の絶対的な値は，コーディングやコーパスにより変わると考えられるが，相対的な速度の順位，つまり，*lcp*法の方が*binsearch*法よりも速いということは，4.4節で述べた計算量の観点から言っても，コーディングやコーパスに影響されないのではないかと考える☆.

5.2.2 スキップされた接尾辞の数

4.3節で述べたように，*lcp*法において，図8における`while`ループによりスキップされる接尾辞の数が非常に大きいときには，線形探索によりスキップするのは不利である．しかし，**図11**と**図12**に示すように，毎日新聞においてもBNCにおいても，スキップされる接尾辞の数は小さいため，線形探索によりスキップしても問題ないと言える．

なお，これら二つの図では，前節と同様に，最長のパターンについてしかスキップされた接尾辞数の平均をグラフとして示していないが，その理由は，前節と同様に，スキップされた接尾辞数の平均がパターン長に依存しないからである．なぜ，依存しないかというと，スキップされた接尾辞数というのは，図1のような接尾辞トライにおいて，先行順に探索したときに，探索が打ち切られたノードにおける子供の数（から1を引いた数）に等しく，かつ，探索の打ち切りが起きる深さが，前節で述べたように，パターン長には依存しない（許容値のみに依存する）からである．

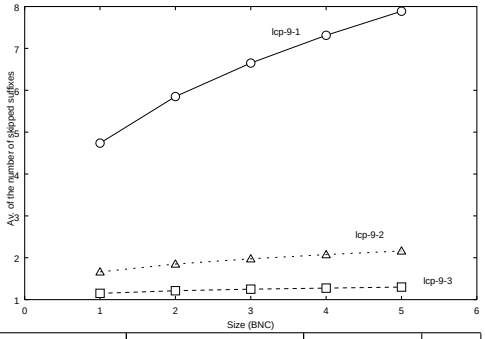
☆ ただし，*lcp*法では，*lcp*配列を索引に使用するため，その分だけのメモリが必要である．そのため，テキストに加えて，接尾辞配列と*lcp*配列とをメモリ上に格納できるような計算機でなければ，*lcp*法での速度向上はないのではないかと予想する．しかし，現在では，数GB 程度のメモリをもつ計算機は普通のものとなってきているので，本稿では，メモリの大きさを変えて速度を比較するということはしなかった．



許容値	2			4		6
長さ	6	12	18	12	18	18
平均スキップ数	12.4	12.4	12.4	2.3	2.3	1.4

図11 毎日新聞でのスキップされた接尾辞数の平均（Size-1〜5のグラフとSize-5における表）.

Fig. 11 Average of the number of skipped suffixes on Mainichi (graphs of Size-1 to 5 and the table of Size-5).



許容値	1			2		3
長さ	3	6	9	6	9	9
平均スキップ数	7.9	8.0	7.9	2.2	2.2	1.3

図12 BNCでのスキップされた接尾辞数の平均（Size-1〜5のグラフとSize-5における表）.

Fig. 12 Average of the number of skipped suffixes on BNC (graphs of Size-1 to 5 and the table of Size-5).

6. 関 連 研 究

6.1 近似文字列照合による全文検索

文献12)では，文字列間の距離を計算するときに，本稿3.2節で説明した動的計画法とは異なる方法を用いている．しかし，彼等も，テキストからパターンを検索するときには，テキストの接尾辞配列を（2分探索により）先行順に走査している．そのため，彼等の距離計算法と，本稿4.2節で説明した接尾辞配列の走査法とを組合せることは可能である．

なお，彼等の距離計算法と，本稿で述べた動的計画法による距離計算法とを比べると，彼等の距離計算法はコスト1の距離に特化した方法であるので，コスト1の距離に関しては，彼等の方法の方がずっと短時間で距離計算が可能である☆．しかし，自然言語処理において，用例間の距離を計算するためには，コスト1の距離だけでなく，もっと一般的な距離を使う必要があると考える．そして，そのような場合でも，動的計画法による距離計算は，適切な距離関数を定義することにより，柔軟に対処できる．そのため，動的計画法により距離計算をすることは有意義であると考える．

6.2 接尾辞配列の走査

接尾辞トライあるいは接尾辞木の後行順の走査を接尾辞配列で（*lcp*配列を利用して）高速に模擬する方法は既に提案されている[8),16)]．ただし，これらの研究においては，接尾辞配列と*lcp*配列とを使う目的は，単語列の頻度などの統計量を得るためである．それに対して，本稿では，先行順の走査を*lcp*配列を利用して模擬する方法を新規に示すとともに，その方法が，近似文字列照合の高速化に有効であることを実証した．

なお，従来の接尾辞配列の走査法[8),16)]と提案手法との主要な違いは，従来の走査法がスタックを利用して接尾辞配列を走査しているのに対して，提案手法はスタックを利用していないことである．ここで，従来の走査法を先行順に修正し，それを利用して近似文字列照合することも可能であると考えるが，我々が4.2節で提案した走査法の方が，本質的な部分が（4.2節で示した）コード1という2重の`for`ループで尽くされることからも分かるように，実装が容易である．

7. お わ り に

本稿では，近似文字列照合による全文検索のための索引としての接尾辞配列を高速に走査する方法を提案するとともに，その有効性を実験により示した．

提案手法は，高速化のために補助的な配列を利用しているが，その配列は，近似文字列照合による全文検索だけでなく，その他にも，単語列の頻度などの統計量を得るのに役立つものである．したがって，提案手法は，その配列の別の有効性を示したとも言える．

電子化されたテキストが大量に蓄積されている現在，自然言語処理が対象とするテキストも大規模になっている．また，自然言語処理だけでなく，言語学，あるいは，外国語学習などにおいても，そのような大規模なテキストにおける言語現象を観察することによる利

☆ その他にも，コスト1の距離については，高速な距離計算法がある[11)]．

益は大きいと考える．

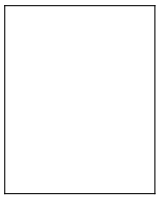
そのような大規模テキストを取り扱うためには，そのための適切なツールが必要である．我々は，そのためのツールを継続的に開発中であり，提案手法も，その一つに組込まれている．我々は，そのようなツールの有効性を，今後，定量的に調べたいと考えている．

参 考 文 献

1) Baeza-Yates, R. and Gonnet, G.: All-Against-All Sequence Matching, Technical report, Department of Computer Science, University of Chile (1990).
2) Baeza-Yates, R. and Gonnet, G.: A Fast Algorithm on Average for All-Against-All Sequence Matching, *Proc. of the 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*, pp. 16–23 (1999).
3) Baldwin, T.: Low-cost, High-performance Translation Retrieval: Dumber is Better, *Proc. of the 39th Annual Meeting and 10th Conference of the Europena Chapter of the Association for Computational Linguistics (ACL-EACL'2001)*, pp. 18–25 (2001).
4) Baldwin, T. and Tanaka, H.: The Effects of Word Order and Segmentation on Translation Retrieval Performance, *Proc. of the 18th International Conference on Computational Linguistics (COLING'2000)*, pp. 35–41 (2000).
5) Cobbs, A.: Fast Approximate Matching using Suffix Trees, *Proc. of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, pp. 41–54 (1995).
6) Gusfield, D.: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press (1997).
7) Ikehara, S., Shirai, S. and Uchino, H.: A Statistical Method for Extracting Uninterrupted and Interrupted Collocations form Very Large Corpora, *Proc. of the 16th International Conference on Computational Linguistics (COLING'96)*, pp. 574–579 (1996).
8) 笠井透, 有村博紀, 有川節夫: 部分語計数問題の接尾辞配列を用いた高速アルゴリズム, 1999年冬のLAシンポジウム (1999). http://www.i.kyushu-u.ac.jp/~arim/jtalks.shtml.
9) Manber, U. and Myers, G.: Suffix Arrays: A new method for on-line string searches, *SIAM Journal on Computing*, Vol. 22, No. 5, pp. 935–948 (1993).
10) Nagao, M. and Mori, S.: A New Method of N-gram Statistics for Large Number of n and Automatic Extraction of Words and Phrases from Large Text Data of Japanese, *Proc. of the 15th International Conference on Computational Linguistics (COLING'94)*, pp. 611–615 (1994).
11) Navarro, G.: A Guided Tour to Approximate String Matching, *ACM Computing Surveys*, Vol. 33, No. 1, pp. 31–88 (2001).
12) Navarro, G. and Baeza-Yates, R.: A Hybrid Indexing Method for Approximate String Matching, *Journal of Discrete Algorithms*, Vol. 1, No. 1, pp. 205–239 (2000).
13) Shang, H. and Merrettal, T. H.: Tries for Approximate String Matching, *IEE Transactions on Knowledge and Data Engineering*, Vol. 8, No. 4, pp. 540–547 (1996).
14) Ukkonen, E.: Finding Approxmate Patterns in Strings, *Journal of Algorithms*, Vol. 6, pp. 132–137 (1985).
15) Ukkonen, E.: Approximate String-Matching over Suffix Trees, *Proc. of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, pp. 228–242 (1993).
16) Yamamoto, M. and Church, K. W.: Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus, *Computational Linguistics*, Vol. 27, No. 1, pp. 1–30 (2001).
17) 長尾真, 石田晴久, 稲垣康善, 田中英彦, 辻井潤一, 所真理雄, 中田育男, 米澤明憲 (編): 岩波情報科学辞典, 岩波書店 (1990).

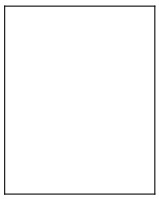
(平成0年0月0日受付)
(平成0年0月0日採録)

（担当編集委員 **通信 太郎**）



内山 将夫（正会員）

1992年筑波大学第三学群情報学類卒業．1997年筑波大学大学院工学研究科博士課程修了．博士（工学）．1997年信州大学工学部電気電子工学科助手．1999年郵政省通信総合研究所非常勤職員．2001年独立行政法人通信総合研究所任期付き研究員．言語処理学会，情報処理学会，ACL，人工知能学会，日本音響学会，各会員．



井佐原　均（正会員）

1978年京都大学工学部電気工学第二学科卒業．1980年同大学院修士課程修了．博士（工学）．1980年通商産業省電子技術総合研究所入所．1995年郵政省通信総合研究所．現在，独立行政法人通信総合研究所けいはんな情報通信融合研究センター自然言語グループリーダー．言語処理学会，情報処理学会，人工知能学会，日本認知科学会，各会員．