Vol.2011-AL-133 No.5
2011/1/12

長いパターンを検出するための文法圧縮に基づく索引構造

岸上直也[†1]　中原昌哉[†1]
丸山史郎[†2]　坂本比呂志[†3]

本研究では，文脈自由文法に基づいた圧縮索引から長いパターンを高速に検索する手法を提案する．提案手法では，$2(1+\varepsilon)n\log n+4n+o(n)$ ビット領域，$O(\frac{1}{\varepsilon}(m\log n+occ_c(\log m\log u)))$ 時間でパターンの出現回数を検出できる．ここで n はサイズ u の原テキストを圧縮し得られた変数の数であり，$m=|P|, 0<\varepsilon<1$ である．実験の結果，ある程度以上の長さを持つパターンについて，LZ-index[8] や FM-index[3] などの既存手法よりも高速に検出できることを確認した．

A data structure based on grammatical compression to detect long pattern

Naoya Kishiue,[†1] Masaya Nakahara,[†1]
Shirou Maruyama[†2] and Hiroshi Sakamoto [†3]

In this research, we propose the method to search long pattern from compressed index based on Context-free Grammar. The proposed method can detect the pattern at $O(\frac{1}{\varepsilon}(m\log n+occ_c(\log m\log u)))$ time with $2(1+\varepsilon)n\log n+4n+o(n)$ bits, where n is generated variables compressed text (original size u), $m=|P|, 0<\varepsilon<1$. Result of experiments, we confirmed our proposed method was faster than existing method (e.g, LZ-index[8], FM-index[3]) at long pattern.

†1 九州工業大学 大学院情報工学府
Kyushu Institute of Technology Graduate School of Computer Science and Systems Engineering
†2 九州大学 大学院システム情報科学府
Kyushu University Graduate School of Information Science and Electrical Engineering
†3 九州工業大学 大学院情報工学研究院
Kyushu Institute of Technology Faculty of Computer Science and Systems Engineering

1. はじめに

本論文では，文法圧縮に基づいた全文検索のための索引構造を提案する．ここで提案する索引構造は，索引自体に原テキストの情報も含んでいる自己索引であり，FM-index[3] や圧縮接尾辞配列[9] のように，索引を構築する際にテキスト中の全ての接尾辞のソートを行う必要がない．そのため，短いパターンの検出は低速であるが，ある程度以上の長さを持つパターンに対しては高速に検出可能である．この特徴を生かし，文書自体をパターンとして用いて文書同士の類似検索を行うといった応用が期待できる．

索引のサイズは，長さ u の文字列を CFG によって長さが 1 になるまで変換し，その時生成された変数の種類を n とすると，$2(1+\varepsilon)n\log n+4n+o(n)$ ビット（ただし，$0<\varepsilon<1$）で表される．また，長さ m のパターン P の出現回数は $O(\frac{1}{\varepsilon}(m\log n+occ_c(\log m\log u)))$ 時間で求められる．ここで occ_c とは，構文木中に現れる P の**コア**の出現回数である．コアとは P の十分長い部分文字列を符号化した変数であり，P が十分長いとき P の出現回数と occ_c はほぼ等しい．

実験では，競合する他の圧縮文字列照合[3),8),9)] との比較を行い，長いパターンにおいて我々の手法が高速に働くことを示す．

2. CFG による圧縮

本節では，原テキストから索引を生成する際の手法について述べる．文字列の置換は至って単純であり，規則を一意に導出するよう制限した文脈自由文法の生成規則で表現する．すなわち，$X\to ab$ という生成規則は a と b を X で置き換える（圧縮する）ことを表し，この置き換えの規則を**辞書**と呼ぶ．これを基本に，**アルファベット縮減法**を用いた ***edit sensitive parsing***[1] によって CFG を生成するのが本論文で提案する圧縮法である．

2.1 アルファベット縮減法

文字の集合 Σ に対して，それらの文字すべてを葉として持つ完全平衡二分木を考え，アルファベット木と呼ぶ．図 1 に $|\Sigma|=11$ の場合のアルファベット木を示す．この木によって，任意の文字の組 (digram) a_ia_j に対して，$lca(i,j)$ を a_i と a_j の最近共通祖先の高さとして定める．文字列 w が与えられ，w の i 番目の文字を $w[i]$ としたとき，w の全ての $w[i]$ について，式 1 によって $w[i]$ の値を $label(w[i])$ に置き換えることをアルファベット縮減と呼ぶ[1]．

ⓒ2011 Information Processing Society of Japan

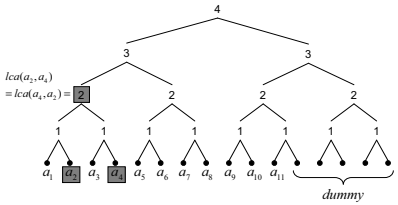

図 1 アルファベット木と lca
Fig. 1 Alphabet tree and lca

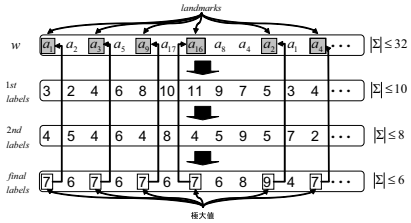

図 2 アルファベット縮減とランドマーク
Fig. 2 Alphabet reduction and landmarks

$$label(w[i]) = \begin{cases} 2 \times lca(w[i-1], w[i]) & (w[i-1] < w[i]) \\ 2 \times lca(w[i-1], w[i]) + 1 & (otherwise) \end{cases} \quad (1)$$

ただし，1 番目の文字は左隣の文字を持たないため，2 番目の文字が 0 番目にあると仮定して変換する．文字 a_i と整数 i を同一視すると，この操作により，$\{1, 2, \ldots, n\}$ で表現された文字列が，$\{2, 3, \ldots, 2 \times \lfloor \log n \rfloor + 1\}$ で構成された同じ長さの文字列に変換される．すなわち，文字の種類が n である文字列にアルファベット縮減を 1 回適用すると，その文字種が $\log n$ に縮減される．そして，文字列 w が連続文字列を含んでいない場合，w に対してアルファベット縮減を適用し得られた文字列 w' もまた連続文字列を含まない．したがって，連続文字列を含まない文字列に対してアルファベット縮減は再帰的に適用可能である．通常，アルファベット縮減を $\log^* n$[⋆1] 回適用することでどのような文字列も，文字の種類が非常に少ない（6 種類程度）文字列に変換することが可能である．そして，アルファベット縮減を $\log^* n$ 適用し得られた文字列の極大値を求め，それぞれの極大値に対応する元の文字列の位置を**ランドマーク**と呼ぶ．例として，文字種が 32 である文字列 w にアルファベット縮減を 3 回適用したときの様子を図 2 に示す．連続文字列を含まない文字列についてアルファベット縮減を適用したとき，以下のことが言える．

定理 1 異なる場所に存在する同じ文字列 w_1, w_2 に対しそれぞれアルファベット縮減を n 回適用し得られた w_1', w_2' は，先頭 n 文字より後ろの文字列が必ず一致する．そして先頭 $2n$ 文字，末尾 $2n$ 文字以外の文字列についてのランドマークも必ず一致する．

このランドマークを用いて，次に説明する *edit sensitive parsing* により圧縮を行う．

⋆1 $\log^* n$ は、$\log \log \cdots \log n > 1$ を満たす log の最大回数を表し，実用的にはほぼ定数と考えてよい．

2.2 Edit sensitive parsing

入力文字列 s を，まずはじめに長さ n の文字列を次の 3 種類の部分文字列に分割する．

(1) 同じ文字が連続する部分文字列
(2) 1 以外の文字列で，長さが $\log^* n$ を超える部分文字列
(3) それ以外の部分文字列

1,3 のタイプの文字列は，左側から 2 文字づつ置き換えていく．もし最後に 1 文字余ったならば，その文字は置換されない．すなわち，文字列 $s_i = aaaab$ が与えられた時，$s_i' = XXb$（$X \to aa$）が得られる．

2 のタイプの文字列については，先ほど説明したアルファベット縮減法によってランドマークを求める．そして，ランドマークとその右隣の文字からなるダイグラムを優先的に置き換え，残りの文字列については左側から 2 文字づつ置き換えていく．文字の置き換えの際，ハッシュ関数などを利用して，既に辞書に登録されているか調べ，登録されていれば同じ変数で置き換えを行う．このようなパージングを *edit sensitive parsing* と言う．例えば，$s_i = abacabcda$ が与えられ，$s_i[2], s_i[6]$ がランドマークであった時，まず $s_i' = aXcaYda$（$X \to ba, Y \to bc$）が得られ，まだ置換されていない部分文字列について左側から 2 文字づつ置き換えることで，最終的に $s_i' = aXZYW$（$X \to ba, Y \to bc, Z \to ca, W \to da$）が得られる．このようにして得られた文字列 $s_1', s_2', \cdots, s_k'$ を連結して s' とし，$|s'| = 1$ となるまで s' を新たな入力文字列 s とし再帰的に圧縮を行い，s に対する構文木を得る．この様子を図 3 に示している．一回のループで文字列がおよそ半分になるため，構文木の高さは高々$\log s$ である．また，この構文木は子の数が必ず 2 つである全二分木である．

3. 圧縮パターン照合

3.1 概　　略

本節では，CFG 圧縮データからの圧縮パターン照合の概略を説明する．テキスト S を圧縮して得られた CFG 圧縮データ G からのパターン検索の基本アイデアは，S を圧縮して得られた辞書集合 D を用いてパターン P を同様に圧縮し，得られた変数列 $P_1', P_2', \cdots, P_k'$ を G の構文木に順序を保って埋め込めるか否かを判定することにある．

定理 1 より，あるテキスト w に現れる部分文字列の異なる出現 $w[i,j] = w[k,l] = \alpha\beta\gamma$ に対して，$w[i,j]$ と $w[k,l]$ のランドマークの位置は β の部分については完全に一致する．ここで $w[i,j]$ は，文字列 w における $w[j]$ までの部分文字列を表す（$i < j$）．これはすなわち，文字列 s がテキスト中のどこに存在していたとしても，s に対するランドマークは両

ⓒ2011 Information Processing Society of Japan

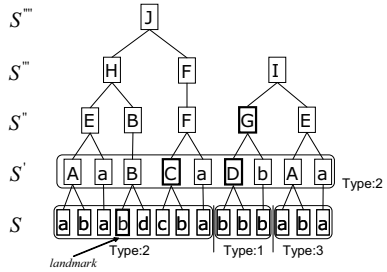

図 3 圧縮の様子
Fig. 3 State of compression

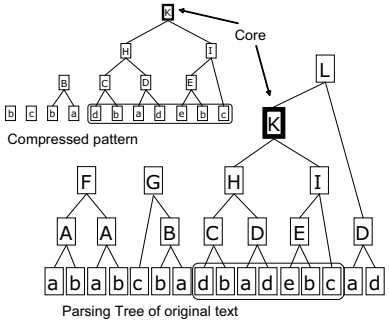

図 4 コアの抽出
Fig. 4 Extracting the Core

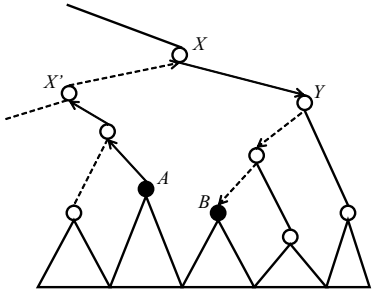

図 5 部分木の隣接関係
Fig. 5 Adjacent relation of subtrees

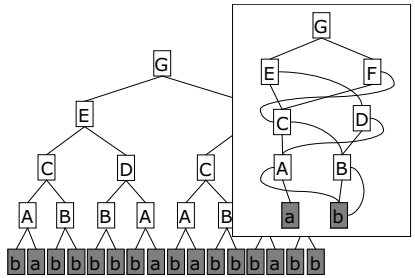

図 6 構文木と DAG
Fig. 6 Parsing Tree and DAG

端の短い不一致部分を除き必ず一致するということである．これにより，以下の性質が成り立つ．

定理 2 $w[i,j] = w[k,l] \in \Sigma_*$ とする．このとき，$w[i,j] = w[k,l] = x\alpha y$ となる十分に長い部分文字列 α が存在して，α が同じ変数に置き換えられる．

もし P が S 中に存在し，その長さが十分長いならば，P の十分長い部分文字列を符号化した共通の変数 X（コア）が変数列 $P'_1, P'_2, \cdots, P'_k$ の中に存在することになり（図 4），その変数が G に現れるかどうかを初めに調べることにより，P の出現を大幅に絞り込むことが出来る．X の出現を絞り込むことが出来たなら，残りの変数列について同じように隣接関係を調べていけば良い．パターンを圧縮して得られた変数列について，隣り合う全ての変数 P'_i,P'_j（$i \neq j$）が構文木上でも隣接していたならば，これをもって P の出現と言い換えることができる．また，定理 1 より，各 P_i は十分長い文字列を符号化しているので，任意の P に対して変数列の数は $k = O(\log |P|)$ となる．

3.2 隣接関係の判定

前節で述べた変数列の隣接関係の判定は，任意の変数 A,B を根とする部分木が構文木内で隣接するか否かを解く問題に帰着できる．まず A の**左辺**，**右辺**，**最右先祖**を以下のように定義する．

定義 1 任意の生成規則（$X \rightarrow ab$）は，構文木上では葉 a,b を持つ根 X として表される．ここで a に対応するものを**左辺**と呼び，b に対応するものを**右辺**と呼ぶ．

定義 2 A の先祖のうち，左辺をちょうど 1 回だけ辿って到達できる A に最も近いノードを A の**最右先祖**と呼ぶ．

直感的には，最右先祖とは自分の右側にある最も近い先祖である．この最右先祖の概念を用いて，ノードの隣接関係を以下のように簡潔に表すことができる．

定理 3 変数 A,B を根とする部分木が構文木内で AB の順番で隣接することと，以下の両方を満たすある変数 X が存在することは等価である．

(1) X は A の最右先祖である
(2) X の右の子（Y）から左辺だけをたどって B に到達可能である

このノードの隣接関係を一般的に表すと図 5 のようになる．最右先祖と同様に**最左先祖**を定義すると，定理 3 は，A,B が隣接するとき A の最右先祖と B の最左先祖は同じ変数であると言い換える事ができる．構文木の高さは入力文字列の長さを u とすると高々$\log u$ であるので，A から最右先祖をたどる操作と B から最左先祖をたどる操作はそれぞれ $\log u$ 時間で行え，2 つのノードの隣接関係は $O(logu)$ 時間で判定できる．

3.3 DAG への変換

構文木上において，任意の変数 A,B の隣接関係を高速に行えることは前節で示した通りだが，一般的に構文木は原テキストよりもサイズが大きい．したがって主記憶上に乗せきれない場合が多く，実用的ではない．そこで，ノードを縮約した DAG を用いた検索を提案する．図 6 は，ある構文木を DAG で表した例である．当然ながら，DAG は同じノードが縮約されているためある変数 A の最右先祖は唯一ではない．このため前節の判定方法を用いることは出来ないが，DAG を 2 つに分解することで DAG 上でも同じように隣接関係を判定することができる．まず，DAG G に仮想シンクを付け加え，G の各ノードの左辺のみからなる G_{left}（左辺木）と右辺のみからなる G_{right}（右辺木）の 2 つの木に分解する．元の

ⓒ2011 Information Processing Society of Japan

Vol.2011-AL-133 No.5
2011/1/12

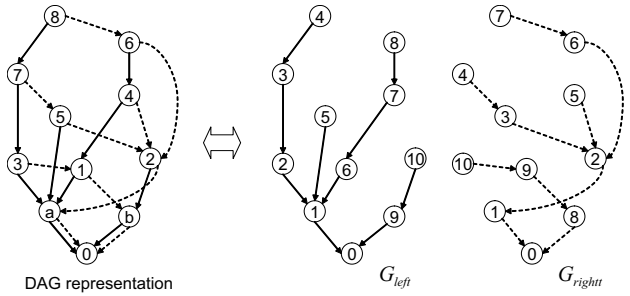

図 7 DAG の分解
Fig. 7 Decomposition of DAG

構文木が全二分木であるので，G の任意のノードは必ず 2 つの子を持つ．したがって，G は必ず一意に左辺木と右辺木に分解できる．この分解の様子を図 7 に示す．

ここで，DAG における変数 A の最右先祖の一つ（X）が確定したと仮定する．G_{left},G_{right} は一番下のノードが根であることに注意すると，構文木における X の右の子供 Y は G_{right} における X の親であり，構文木において Y から B へ左辺のみをたどって到達可能であることは，G_{left} において B が Y の先祖であることと等価である．G_{left} の高さは構文木の高さと同じ $\log u$ であるため，X から B の探索は，DAG 上でも同じく $O(\log u)$ 時間で行える．すなわち，パターン P のコア c に対する最右先祖の一つが定まれば，その最右先祖に対する変数列の隣接関係は $O(\log |P| \log u)$ 時間で行え，コアに対する最右先祖の数は，現実的なテキストにおいては P の出現回数とほぼ一致するするため，全体で $O(occ_c(\log |P| \log u))$ 時間で DAG 上での検索が行える．

3.4 簡潔データ構造を用いた索引構造

ここでは，DAG の分解により得られた木をコンパクトに表現するためのデータ構造と，それらを用いたデータ構造からのパターン検索について述べる．

索引付辞書の簡潔データ構造

アルファベット Σ をもつ文字列 $S[1,n]$ について以下の問い合わせを定義する．

(1) $rank_c(S,i)$:$S[1,i]$ に含まれる $c \in \Sigma$ の数を返す．
(2) $select_c(S,i)$:S 中の c の i 番目の出現位置を返す．
(3) $access(S,i)$:$S[i]$ を返す．

これらの操作を高速に求める一般的な簡潔データ構造として**ウェーブレット木**[4] が知られており，長さ n の文字列に対して $n\log\sigma + o(n\log\sigma)$ 領域のデータ構造ですべての操作を $O(log\sigma)$ 時間で求めることができる．本研究では $|\Sigma| = \sigma = 2$ の場合に，長さ n の文字列に対して，上記 3 つの操作を $O(1)$ 時間で実現する $n + o(n)$ ビット領域の簡潔データ構造[5] を利用する．

順序木の簡潔データ構造

順序木の簡潔データ構造は何種類か存在するが，ここでは括弧列表現（balanced parenthesis representation，以下 BP）[7] を用いる．木 T を順序木とし，T の根ノードの部分木をそれぞれ $T_1, T_2, \ldots, T_d$ と表すと，T に対する括弧列表現 $BP(T)$ は以下のように定義される．

$$\begin{cases} BP(T) = () & (d = 1) \\ BP(T) = (BP(T_1)BP(T_2)\cdots BP(T_d)) & (otherwise) \end{cases}$$

つまり，木の各ノードは開き括弧“(”と閉じ括弧“)”で表され，それぞれ 0,1 に対応付けることでノード数 n の木を $2n$ ビットで表現することが出来る．データ検索を括弧列表現 P を用いて行うには，木の巡回などの操作に対応する操作を P 上で行う必要がある．そのためにまず P 上での基本的な演算を定義する[7]．

(1) $findclose(i)$:$P[i]$ にある開き括弧に対応する閉じ括弧の位置を返す．
(2) $findopen(i)$:$P[i]$ にある閉じ括弧に対応する開き括弧の位置を返す．
(3) $enclose(i)$:$P[i]$ にある開き括弧とそれに対応する閉じ括弧を内部に含むような最小の括弧対の位置を返す．

これらの関数の引数，返り値はあるノードを表す開き括弧の位置であり，これらの操作は定数時間で行える．さらにこれらの操作を用いて新たな操作を定義する．

(1) $parent(x)$:$enclose(x)$
ノード x の親ノードを返す．
(2) $firstchile(x)$:$x + 1$
ノード x の最初の子供を返す．
(3) $nextsibling(x)$:$finclose(x) + 1$
ノード x の次の兄弟ノードを返す．

木の内部ノード，葉ノードはどちらもそれらに対応する開き括弧の位置で表現でき，その位置 i とノードの行きがけ順 p は以下の操作で互いに定数時間で変換できる．

ⓒ2011 Information Processing Society of Japan

Vol.2011-AL-133 No.5
2011/1/12

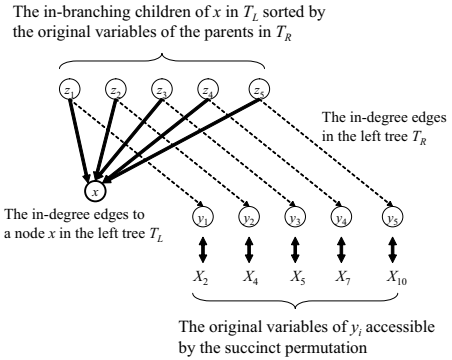

図 8 二分探索による逆引き辞書の実現
Fig. 8 reverse dictionary representation by binary search

$$\begin{cases} p = preorder(i) = rank_{(}(P, i) \\ i = select_{(}(P, p) \end{cases}$$

以上の操作を利用することで，括弧列表現上で木の巡回をシミュレートすることができ，巡回に必要な時間は $O(1)$ であり，$2n + o(n)$ ビットのデータ領域で実現できる[7]．

順列の簡潔データ構造

順列の簡潔データ構造とは，順列 π が与えられたとき，$\pi[i]$ と $\pi^{-1}[i]$ を高速に求めるものである．例えば $\pi = (3, 4, 1, 5, 2)$ の時，$\pi[2] = 4$,$\pi^{-1}[2] = 5$ となる．すなわち，$\pi[i]$ は π の i 番目の要素の値を表し，$\pi^{-1}[i]$ は π 中に i が存在する場所を表す.[6] によると，$(1+\varepsilon)n \log n + O(n)$ ビットの領域を用いれば，$\pi[i]$ を $O(1)$ 時間で，$\pi^{-1}[i]$ を $O(\frac{1}{\varepsilon})$ 時間で計算できることが示されている．

辞書の逆引きのためのデータ構造

パターン P をパージングするために，任意の $z \to xy$ に対して xy から z にアクセスする必要がある．原テキストを圧縮する際にはハッシュ関数などを用いる必要があるが，圧縮して得られた構文木に対する辞書の逆引きは，以下の前処理を行うことで補助データ構造無しに実現することが出来る．$T_L(x), T_R(x)$ をそれぞれオリジナルの変数 x に対応する左辺木・右辺木上でのノードとして，左辺木上での全てのノード $T_L(x_i)$ に対して，$T_L(x_i)$ の子ノード $T_L(z_1), T_L(z_2), \cdots, T_L(z_k)$ を，それぞれ対応する $T_R(z_i)$ の親ノードのオリジナルの変数 y_i の順で並べ替える（図 8）．こうしておくと，xy が与えられたとき，$T_L(x)$ の真ん中の子供（$T_L(k)$）にアクセスし，$T_R(k)$ の親のノードの値が y であれば k を xy の逆引きとして得ることが出来る．もし x と値が違っていれば，その大小によって範囲を狭め、再帰的に $T_L(x)$ の子供を二分探索することで，補助データ構造を必要とせずに $O(logn)$ 時間（n は原テキストを圧縮した時の変数の数）で辞書の逆引きを実現できる．

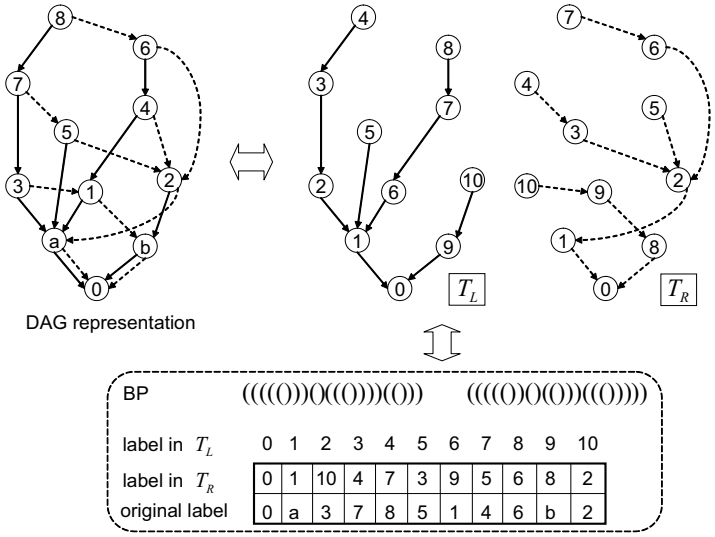

図 9 左・右辺木と簡潔データ構造による表現
Fig. 9 Left/Right tree and succinct representation

以上から，生成された変数の種類を n とすると，前述の左・右辺木をそれぞれ $2n + o(n)$ ビット領域のサイズで表現できる．ただし，BP 表現によりノードのラベルが行きがけ順に書き換えられ，オリジナルの変数名が失われるため，左・右辺木それぞれについてオリジナルの変数名との対応関係を表す配列が必要となる．さらにこれらの配列に対しての逆操作も必要となるため，それぞれの配列に順列の簡潔データ構造を利用する．図 9 に，CFG 圧縮の構文木を簡潔データ構造で表現した様子を示している．このデータ構造上でパターン P を検索するためには，P の長さを m とすると前処理として $O(m \log n)$ 時間でパターンを圧縮し，これにより得られたコアを元に変数列の隣接関係を調べる必要がある．そして，順

ⓒ2011 Information Processing Society of Japan

列の簡潔データ構造を利用しているため木のノードを巡回するには一操作あたり $O(\frac{1}{\varepsilon})$ 時間必要となる．これらをまとめて以下の定理を得る．

定理 4 CFG 圧縮による索引構造は，簡潔データ構造を用いることで $2(1+\varepsilon)n\log n + 4n + o(n)$ ビット領域で表され，$O(\frac{1}{\varepsilon}(m\log n + occ_c(\log m \log u)))$ 時間でパターン P の出現回数を検出することが出来る．

4. 実　　験

本節では，本研究で提案した索引手法と他の圧縮パターン照合との比較実験結果を示す．比較対象の索引構造は，LZ-index[8]，圧縮接尾辞配列（CSArray）[9]，FM-index[3] に基づいた実装である．実験環境は CPU:Intel Xeon E5540(Quad Core, HT @2.53GHz), Memory: 144GB, CentOS 5.5(64bit), gcc 4.1.2 である．我々の手法で用いる順列の完結データ構造のパラメータ ε は $\frac{1}{4}$ とした．

実験では索引のサイズと構築時間，そしてパターンの出現回数を求めるためにかかった時間を計測した．今回実験に用いたテキストデータは Pizza & Chili corpus で公開されている英文テキストデータ 100MB を利用し，先頭からそれぞれ 10,20,40,60,80,100MB 切り取った物である．索引のサイズと構築時間の計測はこれら 6 種類のテキストに対して行った．パターン検出については，100MB のテキストより得られた索引を対象に行い，100MB の原テキストからランダムに 5〜500,000 Byte 切り取ったものをパターンとした．それぞれの長さについて 100 回検出を行いその平均を取っている．ただし LZ-index についてはあまりにも検出時間が長かったため，40,000 Byte で実験を打ち切っている．

図 10 と図 11 はそれぞれの手法の索引の構築時間とサイズを表している．索引サイズは CSArray や FM-index と比べてかなり大きくなってしまったが，構築時間はそれらとほぼ同じである．

次にパターンの出現回数の検索時間の比較を図 12 に示す．この結果から分かるように，本手法は短いパターンの検出に大しては他の手法に比べて非常に遅い．これは短いパターンから得られるコアがテキスト中に頻出するためである．しかし，パターンがある程度長くなってくると，コアの出現頻度とパターンの出現頻度がほぼ一致し，コアの絞り込みが非常に有効に働いていることが分かる．長いパターンにおいては，他の手法の中で一番高速な FM-index と比べても約 1.3 倍ほど速い．

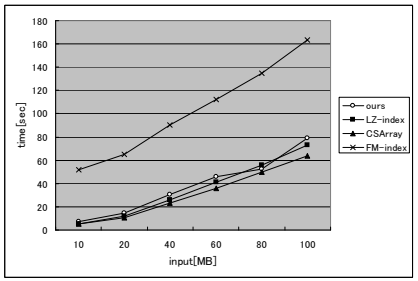

図 10 索引の構築時間
Fig. 10 Time to construct index

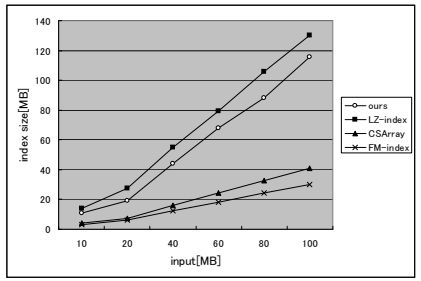

図 11 索引サイズ
Fig. 11 Index size

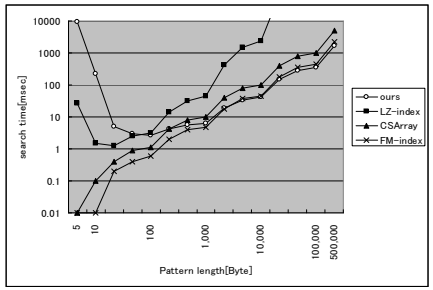

図 12 パターンの出現回数の検出時間
Fig. 12 Time to count occurrences

5. おわりに

本研究では，*edit sensitive parsing* 手法を用いた文法圧縮に基づく索引構造を提案し，理論と実験の両面からその有効性を示した．特に，パターンが長い場合においては一番高速に検索できるという結果が得られた．しかし，索引サイズが原テキストより大きいという問題が残っている．現在の圧縮法では，置き換え後の変数（すなわち，$z \to xy$ の z にあたる部分）は x, y がテキスト中に出現した順に生成されるが，これを x, y の昇順でソートしておくことで，左辺木の幅優先ラベルとオリジナルの変数名が一致する．そして，幅優先にノードを対応付ける括弧列の簡潔データ構造である LOUDS[2] を用いることで左辺木とオリジナルの変数名の対応関係を示す配列が不要となり，領域を削減することができる．予

ⓒ2011 Information Processing Society of Japan

Vol.2011-AL-133 No.5
2011/1/12

備実験を行ったところ，索引サイズをほぼ半分に削減することができ，パターンの検出時間も 20%程度高速化された．これは，左辺木のノードへのアクセスの際にオリジナルの変数名を用いて直接飛べるため，順列の簡潔データ構造にアクセスする回数が減ったからだと考えられる．ただし，辞書の並べ替えを行うことで索引の構築時間が今以上に長くなると予想されるので，より高速に索引を構築できる手法を確立する必要がある．

参 考 文 献

1) Cormode, G. and Muthukrishman, S.: The string edit distance matching problem with moves, *ACM Trans*, Vol.3, No.1 (2010).
2) Delpratt, O., Rahman, N. and Raman, R.: Engineering the LOUDS Succinct Tree Representation, *In WEA2006* (2006).
3) Ferragina, P. and Manzini, G.: Opportunistic data structures with applications, *In FOCS00*, Vol.2, No.1, pp.390–398 (2000).
4) Grossi, R., Gupta, A. and Vitter, J.: High-order entropy-compressed text indexes, *In SODA04*, pp.636–645 (2004).
5) Munro, J.: Tables, *In FSTTCS96*, pp.37–42 (1996).
6) Munro, J., Raman, R., Raman, V. and Rao, S.: Succinct representations of permutations, *In ICALP03*, pp.345–356 (2003).
7) Munro, J. and Raman, V.: Succinct representation of balanced parentheses and static trees, *SIAM Journal on Computing*, Vol.31, No.3, pp.762–776 (2001).
8) Navarro, G.: Indexing text using the ziv-lempel tire, *Journal of Discrete Algorithms*, pp.87–114 (2010).
9) Sadakane, K.: Compressed text databases with efficient query algorithms based on the compressed suffix array, *In ISAAC00*, pp.410–421 (2000).

ⓒ2011 Information Processing Society of Japan