Vol.2011-AL-134 No.9
2011/3/7

文法型圧縮法の全二分木表現による符号化とランダムアクセス手法の提案

丸山史郎[†1]　馬場雅大[†1]
岸上直也[†2]　坂本比呂志[†2]

本稿では文法型圧縮法によって圧縮された文字列上でランダムアクセスを行う手法について述べる．文法型圧縮法とは与えられた文字列を一意に導出する文脈自由文法 (CFG) を構成し，それを符号化することによって実現する圧縮法ある．そのため，全体を復号せずにランダムアクセスを行うためには CFG の構造を保ったまま符号化を行うことが重要となる．本研究では n 個の変数を持つ CFG に対して，$n \log n + 2n + o(n)$ ビット領域で構文木上での操作が可能な符号化法を提案する．この特性とコンパクトな索引付けにより，省主記憶領域で高速なランダムアクセスが実現できることを計算機実験により示す．

Efficient Encoding and Random Access Methods for Grammar-Based Compression

Shirou Maruyama,[†1] Masahiro Baba,[†1]
Naoya Kishiue[†2] and Hiroshi Sakamoto[†2]

In this paper, we present random access methods on a string compressed by grammar-based compression. Grammar-based compression is a well-known technique which constructs a context-free grammar(CFG) G deriving a given string T uniquely and then encodes G. It is important to do the encoding while maintaing the structure of CFGs in order to achieve fast random access query on a compressed string without decoding it. We propose an efficient encoding which is a compressed representation of derivation trees in $n \log n + 2n + o(n)$ bits of space for n variables of G. By this property and compact indexes, we show experimental results that achieve fast random access query in small main memory space.

1. はじめに

文法型圧縮法[7] とは与えられた文字列 T を一意に導出する文脈自由文法 (CFG) を構成し，その文法を符号化する圧縮法であり，LZ 系[18),19)]/Sequitur[14)]/Re-Pair[8)]/Byte Pair Encoding[5)] などの幅広い圧縮法がこの枠組みに属している．本研究は，長さ u の文字列 T を文法型圧縮法により予め圧縮しておくことにより，主記憶上でのランダムアクセスをより省領域で行うことを目的とする．ランダムアクセス問題とは，与えられた整数 $i(1 \leq i \leq u)$ に対して, 文字 $T[i]$ を報告する単純な問題である。通常，$O(u)$ 領域の文字列 T を持っていれば $O(1)$ 時間でこれを行うことができるのは明らかである．しかし，アクセス対象が圧縮データの場合に，多くの圧縮法では一度全体を復元してからアクセスを行わなければならない．本研究では圧縮したデータ全体を復元することなく，できるだけ小さな時間でランダムアクセス可能な圧縮データ構造を提案する．さらにこれを一般化した問題として，与えられた二つの整数 $i, j(1 \leq i \leq j \leq u)$ に対して，部分文字列 $T[i, j]$ を報告する部分文字列復元問題を考える．特にこの問題は近年盛んに行われている圧縮全文索引構造[13)] の基本操作の一つとして示されている．

ランダムアクセス/部分文字列復元可能な圧縮法として，独自の圧縮法，若しくは特定の圧縮法にコンパクトな索引を付加する手法が提案されている．これに対して本研究は文法型圧縮の標準形として知られている直線的プログラム (SLP)[6)] に着目する．SLP とはチョムスキー標準形で表された CFG のサブクラスであり，単元集合を生成する CFG は SLP へ容易に変換することが可能である．よって，本研究の提案手法は文法型圧縮法の枠組みに属する全ての圧縮法に適用できる．SLP 上でのランダムアクセス問題は全ての変数が表す文字列の長さを前もって計算しておくことで $O(n)$ 領域，$O(h)$ 時間で求めることが出来る．また，部分文字列復元問題は $O(h + m)$ 時間で実現できる[4)]．ここで n は CFG の変数の数，h は構文木の最大の高さ，m は復元する部分文字列の長さである．しかし，構文木によっては $h = n$ となる場合があるので，P.Bille et al.[2)] はこれを $O(n)$ 領域，$O(\log u)$ 時間に改善する結果を示している．これらの方法は n が u に対して極端に小さくなる場合に

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

Vol.2011-AL-134 No.9
2011/3/7

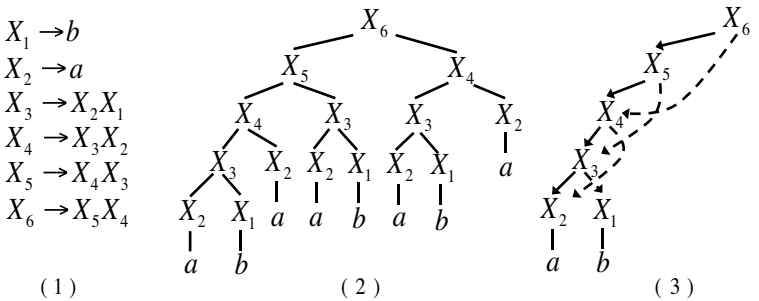

図 1 $T = abaababa$ を導出する CFG $G = (\{X_1, \cdots, X_6\}, \{a, b\}, P, X_6)$ の例．(1) 生成規則 P，(2) 導出の過程を表す構文木表現，(3) 構文木の同一の変数を集約した DAG 表現．DAG 表現の破線は構文木表現での右の子供へ向かう辺を表している．この文法のサイズは 6, 高さは 5 である．

有効であるが、その差が定数倍程度しか変わらない場合には CFG を符号化したまま扱った方が省領域で実現できる．実際に多くの実用的な文法型圧縮法では算術符号などの可変長符号化と組み合わせることで高圧縮率を達成している．しかし，可変長符号上でのランダムアクセスは一度復号しない限り容易ではない．

本研究では CFG で表される構文木の圧縮表現が全二分木であることに着目し，CFG の構造を保ったまま $n \log n + 2n + o(n)$ ビット領域で符号化する手法を提案する．また，馬場らによって提案された全二分木の簡潔データ構造[1] を利用し，$o(n)$ ビット領域の索引を用いることで復号することなく構文木上での基本操作が可能となる．この特性に加えて R.Raman et al. による完全索引付辞書[15] を利用した索引と，復元位置のサンプリングによる索引を用いた二つのランダムアクセス手法を提案し，実験によりその評価を行った．

2. 準　　備

本節では，本稿で必要な予備知識について述べる．なお計算モデルは Word RAM を想定している．

2.1 文　字　列

アルファベット文字の有限集合を Σ とする．Σ 上の全ての文字列集合を Σ^* と表す．**文字列** $T \in \Sigma^*$ の長さを $|T|$ で表す．長さ 2 以上の文字列 $\{a\}^*$ は a の**繰り返し**と呼ぶ．$T[i]$ と $T[i, j]$ は，それぞれ T の i 番目の**文字**と i から j 番目までの**部分文字列**を表している．

2.2 文法型圧縮法

本稿では**文脈自由文法**（Context Free Grammar, CFG）を 4 組 (V, Σ, P, S) で表す．ここで V は変数（非終端記号）の有限集合で $\Sigma \cap V = \Phi$，S は開始記号，P は生成規則の有限集合で V から $(V \cup \Sigma)^*$ への関係である．文法型圧縮法とは，与えられた文字列 T を一意に導出する CFG G を構成する圧縮法である．Charikar et al.[3] によって，そのような最小の CFG を構成する問題は NP-完全であることが知られているため，最適文法の近似解を保証するアルゴリズム[3),16),17)] や貪欲法によって文法を生成するアルゴリズム[8),12)] が提案されている．本稿では，一般性を失うこと無く CFG をチョムスキー標準形として扱う．このような性質を満たす CFG は**直線的プログラム** (Straight Line Program, SLP)[6] と本質的に等価であり，単元集合を生成する CFG は SLP へ容易に変換することが可能である．以下にその SLP の定義を示す．

定義 1. SLP とは以下のように記述された代入文の列である．

$$X_1 = expr_1; X_2 = expr_2; \ldots; X_n = expr_n;$$

ここで X_i は変数，$expr_i$ は単一のアルファベットか $expr_i = X_j \cdot X_k (j, k < i)$ で表される．

つまり，SLP が表す CFG G の全ての変数は生成規則の左辺に一度しか現れない．また，右辺はアルファベット 1 文字か，左辺の変数よりも前に現れた 2 つの変数の連結のどちらかに制限される．この CFG G の変数の数 n を**文法のサイズ**と呼び，$||G||$ と表す．G が表す構文木の最大の高さを**文法の高さ**と呼ぶ．そして，変数 X_i が表す文字列の長さを $|X_i|$ として表す．SLP は導出の過程でループや分岐を含まないため，図 1 のように非巡回有向グラフ (Directed Acyclic Graph, DAG) として表すことができる．この DAG 表現は次節で提案する CFG の符号化方法と深く関係する．

3. 文法の符号化方法

本節では，CFG $G = (V, \Sigma, P, S)$ の符号化方法を示す．最初に以下の手順で構文木の**刈り込み木**を作る．

(1) P から右辺がアルファベット 1 文字の規則を取り除く．
(2) 開始記号 S から左優先順に変数を展開していく．ただし，展開の過程で初めて現れた変数はそのまま展開するが，2 度目以降に現れた変数の展開は行わない．この展開の過程を木構造で表す．
(3) 変数名を表すラベルを内部ノードの前置順で付け直す．

この刈り込み木の例を図 2 に示す．ここで刈り込み木の各内部ノードが持つ変数 Y_i とその

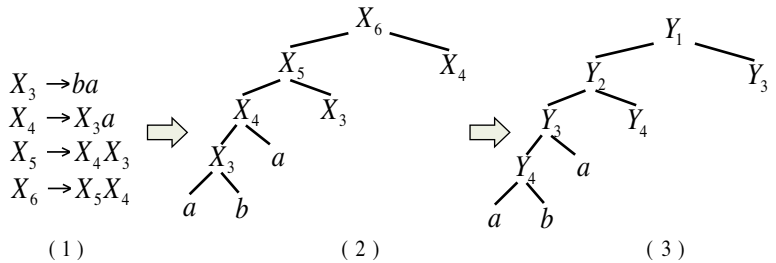

図 2 図 1 の文法から刈り込み木を作る例．(1) P からアルファベット文字を導出する規則を取り除く，(2) 規則を展開しながら刈り込み木を作る，(3) 変数名を付け替える．

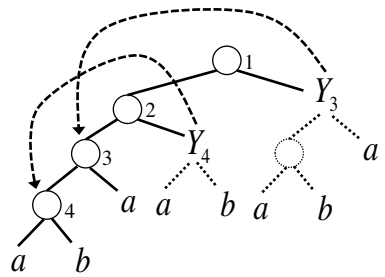

図 3 内部ノードの全置順が変数名に対応する．

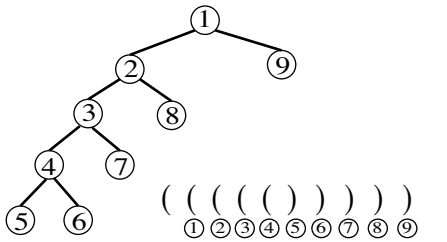

図 4 ラベル無し全二分木の括弧列表現．ノード番号と前置順が対応している．

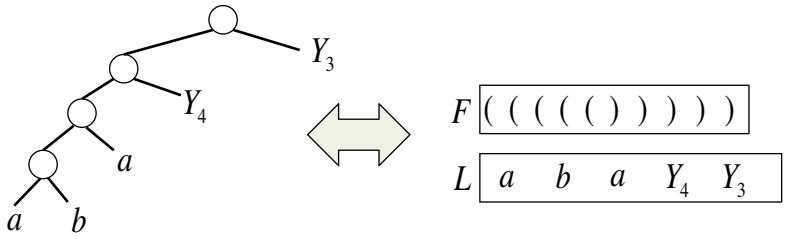

図 5 刈り込み木の符号化表現 (F, L)

左右の子 Y_j, Y_k が生成規則 $Y_i \to Y_j Y_k$ を表していることに着目する．また，葉ノードが持つ変数 Y_i について，Y_i に対応する内部ノードを根とする部分木を再帰的にコピーすることで展開されなかった変数以下の部分木を復元できる．このとき，それぞれの変数名が内部ノードの前置順を表しているので，内部ノードの変数名を陽に持たなくても図 3 のように変数に対応する内部ノードを区別することができる．次にこの刈り込み木を木構造の括弧列表現と葉ノードが持つ記号の列に分解して符号化する．

ラベル無し順序木の括弧列表現方法は様々なものがあるが，刈り込み木は全ての内部ノードがちょうど 2 個の子供を持つ**全二分木**である．H を全二分木とし，$v_1, \ldots v_n$ は前置順での H のノードとする．その時，F によって表される H の括弧列表現は次のように定義される．まず仮想根として $F[0] = '$ ($'$ とする．$1 \le i \le n$ に対して，v_i が内部ノードであれば $F[i] = '$ ($'$ とし，v_i が葉であれば $F[i] = ')'$ とする．この $F[0, n]$ は，長さ $n+1$ の開き括弧と閉じ括弧の対応が取れた括弧列となる．F に仮想根を表す括弧を追加する理由は後に示す木構造の操作に対応するためである．図 4 に括弧列の例を示す．

この括弧列 F に加えて，葉ノードが持つ記号を左から右へ見ていき，配列 L に書きだす．このようにして CFG の刈り込み木は二組 (F, L) として表すことができる．この例を図 5 に示す．(F, L) から元の刈り込み木へは一意に復元できるので，図 3 のように元の文字列に復元可能である．

次の補題によって CFG の符号化表現 (F, L) の領域を見積もる．

補題 1. 任意の SLP が表す CFG $G = (V, \Sigma, P, S)$ は，$n \log n + 2n + o(n)$ ビット領域の二組 (F, L) によって符号化することができる．ただし $\Sigma(|\Sigma| = \sigma)$ は連続した整数アルファベットの集合 $[1, \sigma]$，n は文法サイズ $||G||$ である．

証明. まず，F について考える．$k = n - \sigma$ とすると，刈り込み木の内部ノード数は k，葉ノード数は $k+1$，さらに仮想根を表すノードが 1 つある．また，F は'(' と')' の 2 値アルファベットからなる．よって，F は $2k+2$ ビット領域で表すことができる．次に L について考える．L の長さは葉の数に等しいので，$|L| = k+1$ である．Σ は連続した整数アルファベット $[1, \sigma]$ なので，L のアルファベットは $[1, n]$ で表すことができる．L をそのまま表すと $(k+1)\lceil \log n \rceil$ ビット領域必要になるが，Munro et al. の整数列表現[10] を利用すると $(k+1)\log n + o(k)$ ビット領域で表すことができる．これより二組 (F, L) は $(k+1)\log n + 2k + o(k) \le n \log n + 2n + o(n)$ ビット領域となる． □

もしも Σ が連続した整数アルファベットではない場合は，Σ と $[1, \sigma]$ の対応表を R.Raman et al. の完全索引付辞書[15] によって作れば良い．

本節の最後に (F, L) から元の文字列に戻す方法について簡単に述べる．そのためには (F, L) 上を走査しながら構文木の深さ優先探索を模倣させれば良い．つまり F と L を対応付けながら左から右へ見ていき，生成規則を復元していく．それと同時に刈り込み木の葉

が持つ変数を生成規則を使って順次復号する．この復元アルゴリズムは $O(u)$ 時間，$O(n)$ 領域で動作する．ここで u は元の文字列長，n は文法サイズを表す．これはアルゴリズムが構文木を巡回する時間と等しく，また n 個の生成規則を格納する必要があるからである．(F, L) を工夫して，F 中の $')'$ の後に対応する L 中の記号を埋め込むように符号化を行えばオンラインで復元アルゴリズムを動作させることも可能となる．

4. ランダムアクセス/部分文字列復元方法

本稿では文字列 T に対して，ランダムアクセス/部分文字列問題を以下の関数で表す．

(1) $access(T, i)$ $T[i]$ を返す．

(2) $substring(T, i, m)$ $T[i, i+m-1]$ を返す．

ランダムアクセス問題は 1 文字の部分文字列復元問題であるので，本節では部分文字列問題についての手法を述べる．

Claude and Navarro による CFG に基づく自己索引構造[4] では，SLP 上での部分文字列復元方法を示している．最初にその手法について簡潔に説明する．ただし，彼らのデータ構造はサイズ n の SLP を約 $2n \log n$ ビット領域の二項関係の簡潔データ構造によって表している．そのため，各生成規則にアクセスするために $O(\log n)$ 時間必要となるが，ここでは定数時間でアクセスできるものとして述べる．まず $substring(T, i, m)$ を行うために，最初の位置 i を求める．構文木の根を表す変数 X_n から i 番目の位置を求めるまで再帰的に木を降りていく．規則 $X_i \to X_l X_r$ が与えられたとき，$|X_l| \geq k$ ならば，X_l の方へ降りていく．さもなければ $i := i - |X_l|$ をとして X_r の方へ降りていく．この操作は予め全ての変数の長さを計算しておくことで $O(h)$ 時間で実現できる．ここで，h は文法の高さである．そして，再帰から戻りながら右側の文字列を復元する．もしも左の子からの再帰から戻ったならば，右の子を左から右へ展開する．これを m 文字復元するまで繰り返す．この操作は $O(h+m)$ 時間で実現できる．この理由は，再帰を利用することで辿る辺の数が $O(h+m)$ 本に抑えられるからである．

SLP 上でこの操作を行うためには n 個の規則と変数の長さの情報が必要となる．SLP をそのまま表現すると約 $2n\lceil \log n \rceil$ ビット領域 (Claude and Navarro[4] はこの表現を plain SLP と呼んでいる)，変数の長さの情報は $n\lceil \log u \rceil$ ビット領域が必要となる．本研究では，CFG の符号化表現 (F, L) とコンパクトな索引付け手法によって現実的に省主記憶領域で実現する方法を示す．

4.1 簡潔索引

$\Sigma = \{0, 1\}$ 上のビット列 $B[1, u]$ での基本的な操作について説明する．この B に対して以下の二つの問い合わせが可能なデータ構造を**完全索引付辞書**と呼ぶ．

- $rank_c(S, i)$ $B[1, i]$ に含まれる $c \in \{0, 1\}$ の数を返す．
- $select_c(S, i)$ B 中の $c \in \{0, 1\}$ の i 番目の出現位置 j を返す．

これらの操作について，B に $o(u)$ ビット領域の索引を付加することで定数時間で実現できることが知られている[9]．また 1 を n 個だけ含む長さ u のビット列 $B(u, n)$ について，$\lceil \log \binom{u}{n} \rceil + O(u \log\log u / \log u)$ ビット領域で *access/rank/select* 操作を定数時間で実現できる表現方法が知られている[15]．$H_0(B)$ をビット列 $B(u, n)$ の 0 次経験的エントロピーとすると，$\lceil \log \binom{u}{n} \rceil$ の項は $uH_0(B)$ とほぼ一致する．R.Raman, V.Raman and S.S.Rao によって提案されたこのデータ構造を，本稿では RRR と呼ぶことにする．

次に $\Sigma = \{'(', ')'\}$ から成る括弧列 $F[1, n]$ に対して，*findclose*(F, i)/*findopen*(F, i) を定義する．これは閉じ/開き括弧 $F[i]$ に対応する開き/閉じ括弧 $F[j]$ の位置 j を返す問い合わせである．これらの操作は $o(n)$ ビット領域の索引を使うことで定数時間で実現できる[11]．

4.2 刈り込み木上の基本操作

ここでは符号化表現 (F, L) 上での基本操作方法を示す．まず最初に全二分木の括弧列表現上での簡潔データ構造を説明する．F を前節で述べた全二分木の括弧列表現とする．この F 上での *rank/select*, *findclose/findopen* 操作によって様々な全二分木に対する問い合わせを定数時間で実現できる[1]．ここでは本研究に必要な操作のみを以下に示す．

- *parent*(i) ノード i の親ノードを求める．
  $F[i] =' ('$ ならば $i-1$，そうでなければ *findopen*$(F, i-1)$.
- *left-child*(i) ノード i の左の子を求める．
  $i+1$.
- *right-child*(i) ノード i の右の子を求める．
  findclose$(F, i)+1$.
- *internal-select*(i) 前置順で i 番目の内部ノードを返す．
  $select_(F, i+1)$.
- *leaf-rank*(i) ノード i の左側（自身も含む）にある葉ノードの数を返す．
  $rank_)(F, i)$.
- *leaf-select*(i) 前置順で i 番目の葉ノードを返す．
  $select_)(F, i)$.

Vol.2011-AL-134 No.9
2011/3/7

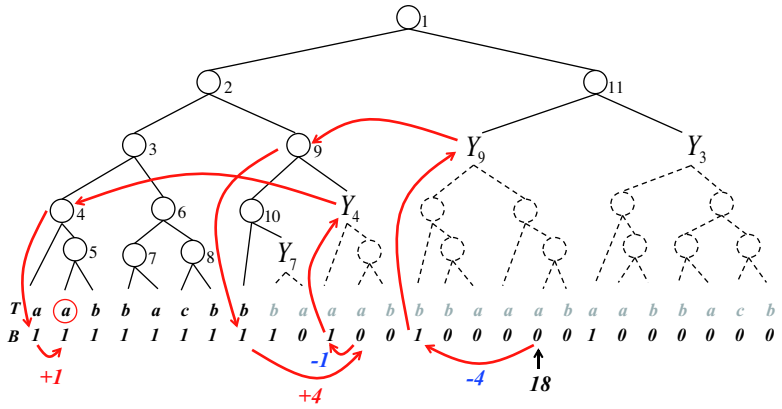

図 6 Algorithm 1 によって $T[18]$ を求める過程．破線は省略されている構文木の部分木を表している．

- *lmost-leaf*(i) ノード i を根とする部分木の最左の葉を返す．
  $F[i] =')'$ ならば i, そうでなければ *select-leaf*(*rank-leaf*$(i)+1$).

以上の問い合わせは各ノードの前置順に対応した整数で行われる．図 4 の例では，*parent*$(7) = 3$ となる．

次に CFG の符号化表現 (F, L) 上での問い合わせを以下に示す．

- *leaf-symbol*(i) 葉ノード i に対応する変数を返す．
  $L[$*leaf-rank*$(i)]$.
- *ref-variable*(Y_i) 変数 Y_i に対応する内部ノードを返す．
  internal-select(i).

これらは定数時間で実現できる．この二つの問い合わせを組み合わせることで，図 3 のように各葉ノードに対応する内部ノードを直接知ることが可能となる．

4.3 RRR による索引

ここでは RRR による部分文字列復元のための索引手法について述べる．文字列 $T[1, u]$ とそれを導出する CFG の符号化表現を (F, L) とする．任意の i 番目の文字 $T[i]$ に対して，k を $i \leq \sum_{j=1}^{k} |L[j]|$ を満たす最小の整数とする．つまり $T[i]$ は $L[k]$ によって符号化されているか $L[k]$ そのものである．この整数 i, k に対して，次のビット列 $B[1, u]$ を定義する．

$$B[i] = \begin{cases} 1 & (i = 1) \\ 1 & (i > 1 \text{ かつ } i = \sum_{j=1}^{k-1} |L[j]| + 1) \\ 0 & (\text{それ以外}) \end{cases} \quad (1)$$

B は長さ u, 高々n 個の 1 を含むビット列であり、これを RRR のデータ構造に変換したものを索引として利用する．

これより (F, L) と索引 B による部分文字列復元アルゴリズムについて述べる．以下の Algorithm 1 は位置 i から始まる長さ m の部分文字列 $T[i, i+m-1]$ を復元するための手続きである．ただし，引数の中の t_s はノードを表す整数であり，初期値は刈り込み木の根を表すノードとなる．

Algorithm 1 Partial-Decode-RRR$((F, L, B), i, m, t_s)$

```
1  b_1 := select_1(B, rank_1(B, i));
2  p := i − b_1;
3  t_1 := leaf-select(rank_1(B, b_1));
4  X := leaf-symbol(t_1);
5  if X ∈ Σ then
6      S := X;
7  else
8      t_2 := ref-variable(X);
9      t_3 := lmost-leaf(t_2);
10     b_2 := select_1(B, leaf-rank(t_3)) + p;
11     S := Partial-Decode-RRR((F, L, B), b_2, m, t_2);
12 end if
13 R := Decode-Right-Leave((F, L), t_s, t_1, m − |S|));
     ▷ ノード t_s を根とする部分木の t_1 より右側の葉を順次復号し，その連結文字列を返す．これは m − |S| 文字復元するか，部分木の最右の葉まで行われる
14 return S · R;
```

図 6 にこの手続きに基づいた先頭文字 $T[i]$ を求める過程を示す．最初に復元開始位置 i を含む葉を求め，葉が変数ならば対応する内部ノード以下の部分木を再帰的に調べることで目的の文字を求める．先頭文字を求めた後は再帰から戻りながら，右側の葉が表す文字列を復元していく．このアルゴリズムはノード t_s が表す文字列に対して，位置 i から右に高々m 文字の文字列を返す関数と考えることができるため，構文木の隣接する m ノードのトップダウン的巡回と同じ時間だけ必要となる．h を文法の高さとすると，辿る必要のある辺数は $O(h+m)$，各辺を辿る時間は全て定数時間なのでこのアルゴリズムは $O(h+m)$ 時間で動作する．これより次の定理が成り立つ．

Vol.2011-AL-134 No.9
2011/3/7

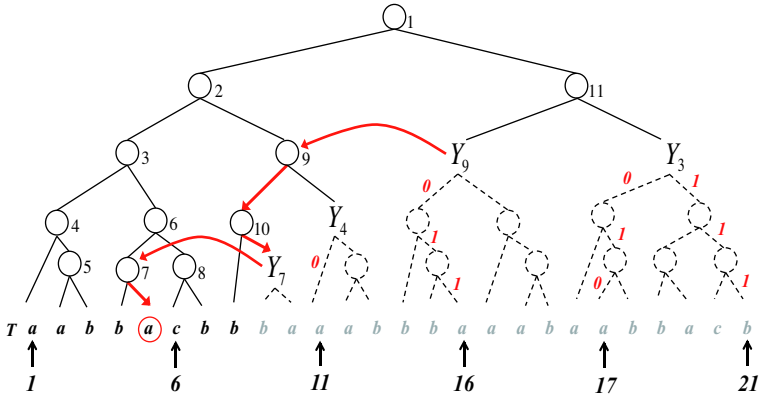

図 7 サンプリング間隔 $s = 5$ の例．復元開始文字 $T[16]$ を求めるためには赤いパスを辿る．

定理 1. CFG の符号化表現 (F, L) に対して，$\lceil \log \binom{u}{n} \rceil + O(u \log\log u / \log u)$ ビット領域の索引を用いることで，任意の部分文字列を $O(h+m)$ 時間で復元できる．ここで n は文法サイズ，u は文字列長，h は文法の高さ，m は復元する部分文字列の長さを表す．

4.4 パスサンプリングによる索引

ここではサンプリングによる索引を使った方法について述べる．これは長さ u の文字列 T の復元開始位置をサンプリング間隔 s 毎に限定することで索引領域をコンパクトにする方法である．任意の整数 i, j に対して部分文字列 $T[i, j]$ を復元したいときは，i に最も近い復元開始位置 k から $T[k, j]$ を復元する．したがって，最悪 s 文字分多くの文字列を復元しなければならない．ここでは 3 つのビット列 B_1, B_2, B_3 による索引を利用する．

ビット列 B_1 は復元開始位置 $p_1, p_2, \cdots p_{\frac{u}{s}}$ が刈り込み木のどの葉ノード以下の部分木に属しているかを示すビット列である．B_1 には'1' が刈り込み木の葉ノード数だけ含まれる．サンプリング位置 p_j が葉ノード t 以下の部分木に存在するときは，t に対応する B_1 中の'1' の直後に'0' を挿入する．図 7 の例では 12 番目の葉に 2 つのサンプリング位置を含むので，12 番目の'1' の後に 2 つの 0 を挿入する．この例では B_1 = "101111101111010100" となる．'1' の数が CFG のサイズ n 以下，'0' の数がサンプリング位置の数 $\frac{u}{s}$ となるので B_1 の長さは高々 $n + \frac{u}{s}$ となる．k 番目のサンプリング位置を含む葉を求めるためには B_1 上で次の操作をすれば良い．

- $\mathit{leaf\text{-}smp}(k) = \mathit{rank}_1(B_1, \mathit{select}_0(B_1, k))$

図 7 の例では左から 11 番目の葉以下の部分木を左 → 右 → 右と辿っていけば 4 番目のサンプリング位置 16 に到達する．この部分木は変数 Y_9 で表されているので，$\mathit{ref\text{-}variable}(Y_9)$ 操作で対応する内部ノード以下の部分木を見れば良い．B_2 は左へのパスを'0'，右へのパスを'1' として葉ノードからサンプリング位置に到達するための全てのパスの連結を表す．ただし，葉の持つ記号がアルファベットの場合もあるので，各パスの先頭に必ず'0' を挿入すると約束する．図 7 の例では，"0", "0", "00", "0011", "0010", "0111" の 6 つのパスが存在し，B_2 = "0000001100100111" となる．B_2 から必要なパスを取り出すためには各パスの区切りを知る必要がある．それをビット列 B_3 で表す．B_2 と同じ長さのビット列を用意し，B_2 の各パスの開始位置に'1'，それ以外を全て'0' とする．図 7 の例では，B_3 = "1110100010001000" となる．B_2, B_3 の長さは文法の高さを h とすると高々 $\frac{hu}{s}$ となる．この B_2, B_3 より k 番目のサンプリング位置へ到達するパスを取り出す操作を以下に示す．

- $\mathit{get\text{-}path}(k) = B_2[\mathit{select}_1(B_3, k), \mathit{select}_1(B_3, k+1) - 1]$

以上のビット列による索引 $B = (B_1, B_2, B_3)$ により，k 番目のサンプリング位置から始まる長さ m の部分文字列を復元するアルゴリズムの概要を示す．

(1) $\mathit{leaf\text{-}smp}(k)$, $\mathit{get\text{-}path}(k)$ により，k 番目のサンプリング位置を含む葉とパスを求める．

(2) パスを先頭から見ていき，0 ならば左，1 ならば右へ辿る．もしも変数を持つ葉に到達したら変数に対応する内部ノードへ移る．これをアルファベット文字を持つ葉に到達するまで再帰的に繰り返す．

(3) Algorithm 1 と同じように，再帰から戻りながら右側の葉が表す文字列を復元する．合計で m 文字分復元したら終了．

このアルゴリズムは $O(h+m)$ 時間で動作する．理由は RRR による索引と同じで，構文木の隣接する m ノードの巡回とほぼ同じ計算時間で抑えられるためである．また，任意の位置 i から復元するためには最も近いサンプリング位置 k から復元する必要があるため，サンプリング間隔を s とすると任意の部分文字列復元時間は $O(h+m+s)$ となる．

これより以下の定理が成り立つ．

定理 2. CFG の符号化表現 (F, L) に対して，$\frac{u}{s}(2h+1) + n + o(\frac{hu}{s} + n)$ ビット領域の索引を用いることで，任意の部分文字列を $O(h+m+s)$ 時間で復元できる．ここで n は CFG の文法サイズ，u は文字列長，h は文法の高さ，m は復元する部分文字列の長さ，s はサンプリング間隔を表す．

表 1 圧縮率（圧縮テキスト長/元テキスト長）[%].

	gzip -9	bzip2 -9	Re-Pair	Plain-SLP	Encoded-SLP
ENGLISH	37.64	28.07	31.79	56.43	30.47
XML	17.12	11.36	16.67	27.21	14.74
SOURCES	22.38	18.67	27.76	46.17	25.01
PITCHES	30.24	34.62	58.23	100.25	54.3
PROTEINS	46.51	44.8	54.15	96.14	51.92
DNA	27.02	25.95	41.78	69.21	37.32

5. 実験結果

本節では，本研究で提案した符号化手法とランダムアクセス/部分文字列復元時間の計算機実験による結果を示す．環境は CPU:2.53GHz Intel Core 2 Duo，メインメモリ:8GB RAM，OS:Mac OS X 10.6.6，実装言語:C 言語，コンパイラ:gcc 4.2.1(-O9 オプション) である．また，実装で使用した簡潔データ構造の一部は Francisco Claude 氏によって公開されている libcds [⋆1]を利用している．

CFG を生成するアルゴリズムは Re-Pair[8] を用いた．これは文字列中の最も頻度の高い隣接する二文字を生成規則によって変数に置き換えていく線形時間アルゴリズムである．Re-Pair では二回以上出現する二文字がなくなると文法の生成を停止するが，ここでは最後にチョムスキー標準形に変換することで SLP を作る．

5.1 圧縮率の測定

本研究で提案した全二分木表現による符号化と他の圧縮法との圧縮率の比較実験結果を表 1 に示す．入力データとして，Pizza & Chili コーパス[⋆2]で公開されている 6 種類のテキスト ENGLISH（200MB），XML（200MB），SOURCES（200MB），PITCHES（56MB），PROTEINS（200MB），DNA（200MB）を利用した．比較する圧縮法として，圧縮プログラムとして広く知られている gzip と bzip2 の-9 オプションの出力， オリジナルの Re-Pair[⋆3]の出力，Re-Pair が生成した CFG を SLP に変換してそのまま出力した Plain-SLP，そして今回提案した SLP の符号化手法 Encoded-SLP の 5 種類である．

実験結果から Encoded-SLP は符号化を行わない Plain-SLP の約半分のサイズになっていることが分かる．また，オリジナルの Re-Pair よりも Encoded-SLP の方が僅かに圧縮率が良い．これは Re-Pair の辞書部分（生成規則）が刈り込み木として埋め込まれることで効率よく符号化出来ているためだと思われる．

⋆1 http://libcds.recoded.cl/
⋆2 http://pizzachili.dcc.uchile.cl/texts.html
⋆3 http://www.cbrc.jp/~rwan/en/restore.html

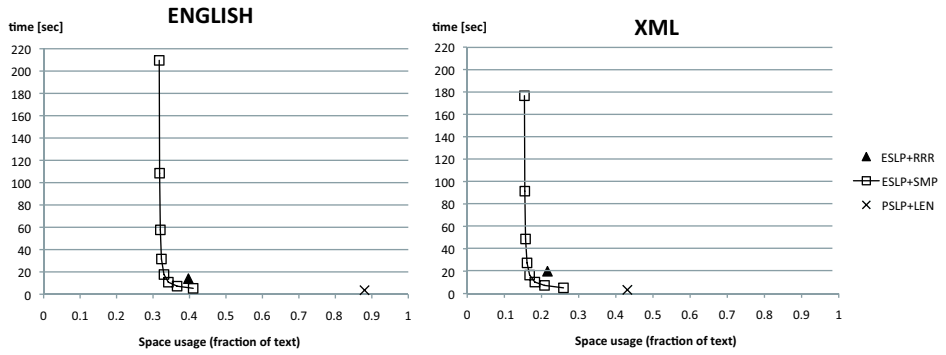

図 8 100 万回のランダムアクセス時間の測定

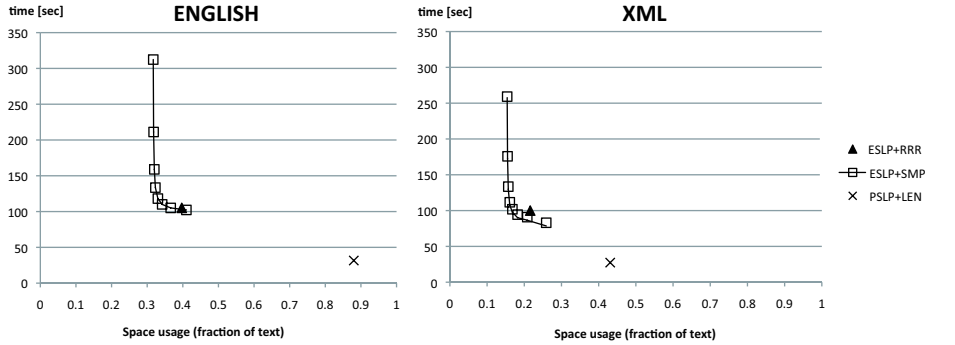

図 9 100 万回の部分文字列（512 文字）復元時間の測定

5.2 ランダムアクセス/部分文字列復元時間の測定

次にランダムアクセスと 512 文字の部分文字列復元時間の測定結果について述べる．実験対象のテキストは ENGLISH（200MB），XML（200MB）の 2 種類である．比較対象は Encoded-SLP に RRR による索引を付けた ESLP+RRR，パスサンプリングによる索引を付けた ESLP+SMP，Plain-SLP に変数の長さ情報を持たせた PSLP+LEN の 3 種類である．ESLP+SMP のサンプリング間隔は 16, 32, 64, 128, 256, 512, 1024, 2048 と変化させ

て測定した．図 8 は 100 万回のランダムアクセス時間，図 9 はランダムな 512 文字復元の測定結果であり，縦軸は時間 [sec]，横軸は元テキスト長に対する使用主記憶領域の割合を表している．ランダムアクセス時間の実験結果で ESLP+RRR がサンプリング間隔の短い ESLP+SMP よりも遅い理由は，RRR のデータ構造を利用した rank/select 操作が一般のビット列上のものよりも数倍遅いためだと思われる．また，PSLP+LEN 上の部分文字列復元時間は提案した二つに比べて約 3 倍速いが，符号化を行っていないため文法サイズに対する領域効率は悪い．提案手法の索引領域は圧縮データ長の 1/3 から無視できる程のサイズとなるので，使用主記憶領域は元テキストの約 15-40%で抑えられている．この実験結果により，現実的なテキストに対して，省主記憶領域でランダムアクセスを実現可能なことが確認できた．

6. おわりに

本研究では，文法型圧縮の枠組みに対して適用出来る効率的な符号化手法を提案した．そして，その符号化の特性を利用したランダムアクセスのための索引付け手法を提案し，実験による評価を行った．本研究の問題点の一つとして，ランダムアクセス時間が構文木の高さ h に依存していることが挙げられる．これは $h = n$ となる場合は最悪問い合わせ時間が $O(n)$ 時間となるためである．この解決策の一つとして，W.Rytter が提案した AVL 文法[16] を応用することで，与えられた CFG の構文木をバランスさせる方法が考えられる．その際，文法サイズが最悪で $O(\log u)$ 倍となることが示されているが，今後は実際にどの程度文法が大きくなるのか検証する必要がある．一方で P.Bille et al.[2] は理論的に SLP へのランダムアクセスを $O(n)$ 領域，$O(\log u)$ 時間に改善しているが，符号化までは考慮されていない．よって，符号化したままの状態で理論的に問い合わせ時間を抑える方法を提案することも今後の課題である．

参考文献

1) M.Baba, H.Ono, K.Sadakane, and M.Yamashita. A succinct representation of a full binary tree (in Japanese). *IPSJ SIG Technical Report*, 2010-AL-129(1):1–8, 2010.
2) P.Bille, G.M. Landau, R.Raman, K.Sadakane, S.R. Satti, and O.Weimann. Random access to grammar-compressed strings. In *Proc. ACM-SIAM SODA*, 2011.
3) M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
4) F.Claude and G.Navarro. Self-indexed text compression using straight-line programs. In *Proc. 34th Mathematical Foundations of Computer Science*, pages 235–246, 2009. to appear in Fundamenta Informaticae.
5) P.Gage. A new algorithm for data compression. *The C users Journal*, 12(2), 1994.
6) M.Karpinski, W.Rytter, and A.Shinohara. An efficient pattern matching algorithm strings with short descriptions. *Nord. J. Comput*, 4(2):172–186, 1997.
7) J.C. Kieffer and E.-H. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.
8) N.J. Larsson and A.Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000. Announced at DCC 1999.
9) J.I. Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science(FSTTCS)*, pages 37–42, 1996.
10) J.I. Munro, R.Raman, V.Raman, and S.S. Rao. Succinct representations of permutations. In *Proc. of the 30th International Colloquium on Automata, Languages, and Programming(ICALP)*, pages 345–356, 2003.
11) J.I. Munro and V.Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
12) R.Nakamura, S.Inenaga, H.Bannai, T.Funamoto, M.Takeda, and A.Shinohara. Linear-time off-line text compression by longest-first substitution. *Algorithms*, 2(4):1429–1448, 2009.
13) G.Navarro and V.Makinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
14) C.G. Nevill-Manning and I.H. Witten. Identifying hierarchical strcture in sequences: A linear-time algorithm. *J. Artif. Intell. Res.*, 7:67–82, 1997.
15) R.Raman, V.Raman, and S.S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. ACM-SIAM SODA*, pages 233–242, 2002.
16) W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
17) H.Sakamoto, S.Maruyama, T.Kida, and S.Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Trans. on Information and Systems*, E92-D(2):158–165, 2009.
18) J.Ziv and A.Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
19) J.Ziv and A.Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.