Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

効率よいVF符号のための MDL原理に基づく分節木の訓練手法

吉田 諭史[†1] 喜田 拓也[†1]

本稿では，VF符号における辞書データ構造を訓練することでVF符号の性能を向上させる方法について論じる．ここで議論するVF符号とは，分節木と呼ばれる木構造を用いて入力テキストを可変長のブロックに分割し，各ブロックに固定長の符号語を割り当てることでテキスト圧縮を達成する情報源符号化方法である．VF符号の圧縮率は分節木によって決定されるが，入力テキストに対して最適な分節木を構築することはNP困難であることが知られている．そのため，最適に近い分節木を構築するために，入力テキストを繰り返し走査しつつ分節木を訓練させるという手法が提案されている．しかしながら，既存の手法では，符号化の際に事前に符号語長をパラメータとして与える必要があった．符号語長を長くすればテキストの分割数は減少するが，逆に分節木を保存するコストが増大し最終的なデータ圧縮率を低下させうる．本稿では，MDL原理に基づく指標を示し，それによって分節木に登録する文字列を貪欲に決定していく訓練アルゴリズムを提案する．このアルゴリズムは，訓練する過程において符号語長を自動的に調節する．

A method of training parse trees for efficient VF coding based on MDL principle

Satoshi Yoshida[†1] and Takuya Kida[†1]

In this paper, we discuss a method to improve the performance of VF codes by training the dictionary data structures. A VF code, we discuss here, is a source coding scheme that parses an input text into variable-length blocks with a dictionary tree, which is called a parse tree, and then encodes them by fixed-length codewords. Although the compression ratio of a VF code depends on the parse tree, the problem of constructing the optimal tree for an input text is NP-hard. Thus, the methods that train the parse tree by scanning the text repeatedly in order to reconstruct the tree so that it closes to the optimal one, have been proposed so far. However, the existent algorithms need the codeword length as a parameter before encoding the input. Although the number of parsed blocks decreases when we set the codeword length long, the compression ratio can be depressed since the cost for storing the parse tree increases. In this paper, we present a criterion based on the MDL principle and propose a training algorithm that greedily determines strings to be entered into the parse tree according to the criterion. The proposed algorithm automatically adjusts the codeword length while the training.

1. はじめに

テキスト圧縮は，テキスト中に含まれる冗長性をコンパクトに表現することで，記憶のための領域を削減する技術である．これは主に，歪のない情報源符号化を用いて実現される．今日までに，Huffman符号やRun-length法，LZ系圧縮法など，数多くの圧縮手法が提案されており，今なお盛んに研究されている[6),8)]．

歪のない情報源符号化において，符号とは，情報源アルファベット上の任意の記号列を符号アルファベット上の記号列に写す1対1写像に他ならない．このような観点からみると，符号化手法は次の4種類に大別できる．

FF符号 (fixed-length-to-fixed-length code): ある一定の長さ L ごとに情報源系列を分割し，それぞれを固定長 ℓ の符号語へと変換する符号化．

FV符号 (fixed-length-to-variable-length code): ある一定の長さ L ごとに情報源系列を分割し，それぞれを可変長の符号語へと変換する符号化．

VF符号 (variable-length-to-fixed-length code): 可変の長さに情報源系列を分割し，それぞれを固定長 ℓ の符号語へと変換する符号化．

VV符号 (variable-length-to-variable-length code): 可変の長さに情報源系列を分割し，それぞれを可変長の符号語へと変換する符号化．

良く知られたHuffman符号は，固定長の記号列（1文字ごと）に可変長の符号語を割り当てるFV符号である．また，LZ系圧縮法などはVV符号とみることができる．

多くの場合において，テキスト圧縮で重要視される要素は圧縮率である．しかしながら，VF符号は符号語長が固定長であるという制約から，高い圧縮率を達成することが難しい．そのため，現在，可変長符号化であるFV符号やVV符号の研究が主流である[7)]．しかし，近年，テキスト圧縮を利用してパターン照合処理を高速化するという視点から，固定長符

†1 北海道大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Hokkaido University

Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

号である VF 符号が見直されている[4),5)]．また，符号語長が固定であることから，任意の位置から圧縮テキストを復号することも容易であり，さらに部分的な再圧縮にも都合が良いなど，VF 符号は工学的に多くの利点を持っている．このような背景から，高い圧縮率を達成できる VF 符号の開発が望まれている．

古典的な VF 符号である Tunstall 符号[10)] は，Huffman 符号同様，記憶のない情報源に対してエントロピー符号であることが証明されており，極限では情報源のエントロピーにまで 1 文字あたりの平均符号長が漸近する．しかしながら，実際のところ，その漸近する速度は符号語長に対して非常に緩やかであり，現実的には Huffman 符号ほどの圧縮率を得ることは難しい[16)]．これまでに，比較的少数ではあるものの，VF 符号を改善する手法がいくつか提案されている．

Tjalkens と Willems ら[9)] は，マルコフ情報源に対する VF 符号の研究に取り組み，実用的な符号・復号化の実装方法を示した．また，Ziv[15)] は，マルコフ情報源に対する VF 符号が FV 符号よりも早くエントロピーに漸近することを証明した．1997 年までの FV 符号および VF 符号については，網羅的な調査が Abrahams によってなされている[1)]．また，Visweswariah ら[12)] は，辞書を用いない VF 符号の性能について報告している．Yamamoto と Yokoo ら[13)] は，Tunstall 符号を元に，複数の分節木を切り替えながらテキストを分割する AIVF 符号 (Almost Instantaneous VF code) を提案した．この AIVF 符号は，Huffman 符号並の圧縮率を達成できることが確認されている[14)]．また近年，Klein，Shapira ら[5)] と Kida[4)] は，入力テキストに対する接尾辞木を短く刈り込んだ木を分節木として用いる VF 符号を，各々独自に提案している[⋆1]．これらは，特に自然言語テキストに対して高い圧縮率を得ることができる[16)]．

VF 符号の圧縮率は分節木の質と大きさによって決まる．前者については，与えられたテキストに対して，ある一定数の符号語を割り当てる分節木のうち最適なものを得るという問題の困難さが指摘されており，NP 困難であることが知られている[5)]．一方，分節木の大きさは符号語長に依存する．符号語長を長くすれば指数的に分節木が大きくなり，より長い文字列を一つの符号語で表現できるようになる．よって，符号語長を長くすればテキストの分割数は少なくなる．しかしながら，一般に VF 符号では復号時にも分節木の情報が必要であるため，逆に分節木を保存するコストが増大し最終的なデータ圧縮率が低下してしまうことがある．また，巨大な分節木を構築するためには多くの時間とメモリ容量を必要とするため，実用的な観点からは分節木を際限なく大きくすることはできない．

そこで，我々はこれまでに，与えられた符号語長と入力テキストに対して，繰り返しテキストを走査して分節木をより良いものに再構築する貪欲な訓練手法を提案している[11)]．この訓練手法によって，符号語長を 16 としたときの訓練で，gzip 以上の圧縮率を達成できる分節木が構築できることが分かった．ただし，あらかじめ符号語長を与える必要があることや圧縮に多大な時間がかかるなど，解決すべき課題点が残っている．

本稿では，MDL 原理に基づいて分節木に含める文字列を貪欲に決定していく訓練アルゴリズムを提案する．すなわち，提案アルゴリズムは，分節木を再構築していく際に，符号化されたブロック系列の長さの総和と分節木を符号化したものとの和が短くなるかどうかを基準に，登録する文字列の有用性を判断する．また，同じ基準でもって自動的に符号語長も調節するため，あらかじめ設定するパラメータが不要になる．提案アルゴリズムは，入力テキスト長 N とすると，$O(N)$ 領域を使って $O(rN^2)$ 時間で動作する．ここで，r は訓練中のテキスト走査の回数であり，入力テキストを初期木で分割したときの分割数を M とすると $O(M)$ で抑えられる数である．

2. 準　　備

集合 Σ を**有限アルファベット**とする．Σ の要素を**文字**とよぶ．Σ^* は Σ 上の文字列すべてからなる集合である．集合 S の大きさを $|S|$ とかく．よって，アルファベット Σ の大きさは $|\Sigma|$ とかくことができ，これはアルファベットに含まれる文字の種類の数を表す．また，文字列 $x \in \Sigma^*$ の長さを $|x|$ とかく．特に，長さが 0 の文字列を**空語**とよび，ε で表す．したがって，$|\varepsilon| = 0$ である．また，$\Sigma^+ = \Sigma^* - \{\varepsilon\}$ と定義する．二つの文字列 x_1 と x_2 を連結した文字列を $x_1 \cdot x_2$ で表す．特に混乱がない場合は，これを x_1x_2 と略記する．また，$w, x, y, z \in \Sigma^*$ について，$w = xyz$ が成り立つとき，x, y, z をそれぞれ w の**接頭辞**，**部分文字列**，**接尾辞**とよぶ．すべての接頭辞や接尾辞は部分文字列でもあることに注意する．文字列 w の i 番目の文字を $w[i]$ と表し，i 番目から j 番目までの文字を連続して並べた部分文字列を $w[i:j]$ と表す．すなわち，$w[1:j]$ は w の接頭辞であり，$w[i:|w|]$ は接尾辞である．便宜的に，$i > j$ の場合は $w[i:j] = \varepsilon$ と定義する．

木構造のうち，子のあるノードを**内部ノード**，子のないノードを**葉ノード**（または**葉**）とよぶ．親を持たないノード（すなわち木の頂点）を**根**あるいは**ルートノード**とよぶ．

⋆1 前者[5)] は，刈込みを行う際に，接尾辞木を深さ優先探索する DynC(Dynamic Cut) というアルゴリズムを提案しているのに対し，後者[4)] は幅優先的に木のノードを選択していくアルゴリズムを提案している．後者を特に STVF(Suffix Tree based VF) 符号と呼ぶが，本質的には DynC による符号と同じものである．

Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

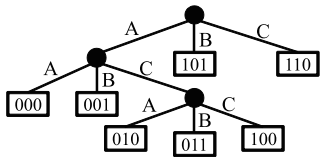

図 1 分節木の例

3. VF 符 号

以下に，VF 符号の符号化方法について概説する．先に述べたとおり，VF 符号とは，情報源系列の可変長の部分系列に対して固定長の符号語を割り当てる符号のことである．ただし，Tunstall 符号のように分節木を用いて符号化する方式が最も基本的かつ汎用的であるので，以降では特にこの形式の VF 符号について説明する．

入力テキスト $T \in \Sigma^+$ に対して，これを長さ $\ell \geq 1$ ビットの符号語で VF 符号化する場合を考える．いま，L 個の葉をもつ分節木 $\mathcal{T}$ が与えられているとする．$\mathcal{T}$ の各葉は，ℓ ビットの整数で番号付けされている．ただし，$L \leq 2^\ell$ である．このとき，$\mathcal{T}$ によるテキストの符号化は，以下の手順で行われる．

(1) $\mathcal{T}$ の根を探索のスタート地点とする．
(2) 入力テキストから記号を 1 個読み取り，分節木 $\mathcal{T}$ 上の現節点からその記号でラベル付けされた子へと移る．もし，葉に到達したら，その葉の番号を符号語として出力し，探索の地点を根へ戻す．
(3) ステップ 2 をテキストの終端まで繰り返す．

例えば，図 1 の分節木でテキスト $T := AAABBACB$ を符号化すると，符号語の系列は 000/001/101/011 となる．分節木によって分割された各部分文字列を**ブロック**とよぶ．たとえば，この例において，符号語 011 はブロック ACB を表現している．

VF 符号では，復号時に元の分節木を復元する必要があるため，復元に要する情報を保持しておく必要がある．Tunstall 符号の場合は，Huffman 符号と同様に，文字の頻度表だけから分節木を復元できる．しかしながら，一般には分節木全体を符号化してテキストの符号化系列と共に保存しなくてはならない．一旦，分節木を復元してしまえば，VF 符号の復号は簡単である．符号系列を ℓ ビットごとに区切って読み込み，切り出された符号語に対応する文字列を分節木を参照しながら出力すればよい．

分節木の符号化は，木構造と辺上のラベルに分割して考える．一般の木の簡潔な表現方法については既に効率良い手法が提案されている[2),3)] が，ここでは単純に括弧列 (balanced parentheses sequences) による符号化を行うとする．この場合，分節木のノード数 N に対して，$2N$ ビットで符号化できる．また，各ノードに対して，符号語をもつかどうかの情報をもたせる必要がある．この情報は 1 ビットで表すことができるので，N ビット追加で必要となる．辺上のラベルについては，分節木を前順走査した順番に並べて保存する．図 1 の例では各ラベルには 1 文字しか割り当てられていないが，接尾辞木のように辺に割り当てられるラベルが空でない文字列であるような分節木もありうる．したがって，ラベル同士の間にはアルファベット上にない区切り文字を挟む必要があるので，一文字あたり $\lceil \log(|\Sigma|+1) \rceil$ ビットで表現されることになる．ラベルの長さの総和を W とすると，ラベル列の保存には $(W+N-1)\lceil \log(|\Sigma|+1) \rceil$ ビットが必要となる[⋆1]．一方，テキストの符号化系列は，分割数を P とすると ℓP ビットで表現できることは明らかである．以上の議論から，最終的に出力される圧縮テキストのサイズは，

$$\ell P + 3m + (W+N-1)\lceil \log(|\Sigma|+1) \rceil \quad \text{(ビット)} \tag{1}$$

となる．すなわち，入力テキストに対して，この式 (1) を可能な限り小さくするような分節木を構築することが目標となる．

次に，文献[11)] で示されている分節木の訓練手法について概説する．この手法は，分節木に登録されている各文字列について，テキスト分割に実際に使用されている回数をカウントし，非頻出な文字列を，別のより使用が期待できる文字列と置き換えることで分節木を再構築していく．具体的には次のとおりである．与えられるテキストを S，構築済みの分節木を $\mathcal{T}$ とする．いま，テキスト S が分節木 $\mathcal{T}$ によって位置 i から j で分割されたとする．すなわち，文字列 $S[i:j]$ は $\mathcal{T}$ に登録されている文字列のうち，S の位置 i から一致する最長の文字列であったとする．このとき，$\mathcal{T}$ 中の $S[i:j]$ はテキスト分割に使われたので，$S[i:j]$ の**受理カウント**(acceptance count)$A(S[i:j])$ を 1 増やす．一方で，$S[i:j+1]$ は $\mathcal{T}$ に含まれていなかったが，もし含まれていなたなら平均ブロック長が伸びたかもしれないので，これを有望な文字列と考え，$S[i:j+1]$ の**期待カウント**(favorable count)$F(S[i:j+1])$ を 1 増やす．テキスト分割が終了した時点で，S の任意の部分文字列 $x, y (x \neq y)$ について $A(x)$ と $F(y)$ を比較し，$A(x)$ が低い x から順番に $\mathcal{T}$ から取り除き，代わりに $F(y)$ が高かった y を $\mathcal{T}$ へと登録する．この手順を，置換えが起こらなくならない限り，ある規定の

⋆1 もちろん，ラベル系列にさらなる符号化を施すことで保存コストを削減することは可能である．ここでは，解析の簡便のため，単純な方法で保存することにしている．

Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

回数まで繰り返す.

以上の訓練手法を用いると，符号語長 16 ビットの Tunstall 符号の分節木（Tunstall 木）が，およそ 20 回程度の繰り返しで gzip 並の圧縮率をもたらすものへと組み替えられる．ただし，圧縮時間は繰り返した回数に比例して増大する．

4. 提 案 手 法

式 (1) を最小にするような分節木を構築できれば最適な VF 符号が得られる．しかし，最適な分節木を構築することは第 1 節で述べたとおり NP 困難であることが知られているため，現実には難しい．そこで，本節では，分節木の記述長と分節木を用いて表されたデータの長さの和である式 (1) を貪欲手法を用いて小さくすることにより，より高い圧縮率を得る分節木を構築するアルゴリズムを提案する．提案アルゴリズムを図 2 に示す．このアルゴリズムの基本的なアイデアは，入力テキストの先頭から順次，隣接するブロックを連結し，式 (1) が小さくなるならば，連結を適用し，そうでなければその部分を連結せずに，次の 2 つのブロックの連結を試みることである．

ここで，提案アルゴリズムの詳細を述べる．このアルゴリズムの入力は，テキスト S であり，出力は，分節木 $\mathcal{T}$ である．まず，初期木として，ルートノードとその $|\Sigma|$ 個の子からなる木を構築し，これを $\mathcal{T}$ とする．なお，ルートノードの各子は Σ の要素でラベルづけされているものとする．次に，$f_{\mathrm{MDL}}(S,\mathcal{T})$ を，$\mathcal{T}$ を用いて S 符号化したときの式 (1) の値を返す関数とする．この関数の計算には，符号語長や分割数も必要であることに注意する．まず，初期木 $\mathcal{T}_0$ に対する $f_{\mathrm{MDL}}(S,\mathcal{T}_0)$ の値を A に代入する．次に，分節木が変化しなくなるまで以下を繰り返す．すなわち，p_1 を入力テキストの位置 i から始まる $\mathcal{T}$ で分割されたブロックとし，p_2 を p_1 の次のブロックとする（図 3）．木 $\mathcal{T}'$ を $\mathcal{T}$ のコピーとし，ブロック p_1p_2 を $\mathcal{T}'$ に追加する．そして，$\mathcal{T}'$ に対する $f_{\mathrm{MDL}}(S,\mathcal{T}')$ の値を A' に代入する．もし，$A' < A$ である，すなわち，ブロック p_1p_2 を追加した場合に式 (1) の値が小さくなるならば，$\mathcal{T} \leftarrow \mathcal{T}'$ として，ブロック p_1p_2 の追加を受け入れる．そして，テキストの位置 i で始まるブロック（このときには p_1p_2 となっていることに注意）とその次のブロックとを連結することを試みる（図 4）．そうでない場合，すなわち，ブロック p_1p_2 を追加した場合に式 (1) の値が小さくならないのであれば，ブロック p_1p_2 の追加はなかったことにする．そして，テキストの位置 i を p_1 だけ進めて，ブロック p_2 とその次のブロックとを連結することを試みる（図 5）.

次に，このアルゴリズムの計算量について述べる．関数 f_{MDL} の計算は，入力テキスト

Algorithm TrainingParseTree
Input: テキスト $S := S[1]\dots S[N]$
Output: 分節木 $\mathcal{T}$

1: $\mathcal{T} \leftarrow$ ルートノードと Σ の要素でラベル付けされた $|\Sigma|$ 個の子からなる初期木 $\mathcal{T}_0$
2: $\mathcal{T}' \leftarrow \mathcal{T}$
3: $A \leftarrow f_{\mathrm{MDL}}(S,\mathcal{T})$
4: **do**
5: $\quad i \leftarrow 0$
6: $\quad$ *trained* $\leftarrow$ **false**
7: $\quad$ **while** $i < N$ **do**
8: $\quad\quad p_1 \leftarrow S[i:N]$ の接頭辞でなおかつ $\mathcal{T}$ に登録された最長の文字列
9: $\quad\quad p_2 \leftarrow S[i+|p_1|:N]$ の接頭辞でなおかつ $\mathcal{T}$ に登録された最長の文字列
10: $\quad\quad \mathcal{T}' \leftarrow \mathcal{T}$
11: $\quad\quad \mathcal{T}'$ に文字列 p_1p_2 を登録する
12: $\quad\quad A' \leftarrow f_{\mathrm{MDL}}(S,\mathcal{T}')$
13: $\quad\quad$ **if** $A' < A$ **then**
14: $\quad\quad\quad \mathcal{T} \leftarrow \mathcal{T}'$
15: $\quad\quad\quad A \leftarrow A'$
16: $\quad\quad\quad$ *trained* $\leftarrow$ **true**
17: $\quad\quad$ **else**
18: $\quad\quad\quad i \leftarrow i + |p_1|$
19: $\quad\quad$ **end if**
20: $\quad$ **end while**
21: **while** *trained* = **true**
22: **return** $\mathcal{T}$

図 2 MDL 原理に基づいて分節木に含める文字列を貪欲に決定する訓練アルゴリズム

Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

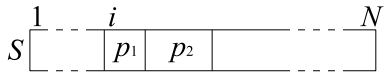

図 3 ブロック p_1 と p_2 の関係

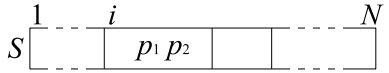

図 4 ブロック p_1 と p_2 を連結した

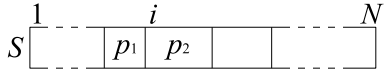

図 5 ブロック p_1 と p_2 を連結せずにテキストの位置を進める

を 1 回走査する必要があるため，$O(N)$ 時間かかる．そのため，1 行目から 3 行目までは $O(N)$ 時間かかる．7 行目から 20 行目までの **while** ループは最大で N 回反復される．ここからは **while** ループ内について考察する．8 行目と 9 行目はそれぞれ $O(|p_1|)$ と $O(|p_2|)$ 時間で実行できる．また，分節木の大きさは $O(N)$ なので，10 行目は $O(N)$ 時間かかる．11 行目と 12 行目はそれぞれ $O(p_1p_2)$ 時間と $O(N)$ 時間かかる．2 つのブロック p_1 と p_2 について，$|p_1|, |p_2| < N$ が成り立つことを考えると，**while** ループは全体で $O(N^2)$ 時間かかることがわかる．したがって，4 行目から 21 行目までの **do-while** ループの反復回数（これは訓練のためにテキストを走査した回数である）を r とすると，このアルゴリズムは全体で $O(rN^2)$ 時間かかることがわかる．また，各ブロックの大きさや，分節木の大きさを $O(N)$ で抑えることができるので，このアルゴリズムの領域計算量は，$O(N)$ である．

ここで，アルゴリズムの計算時間の改良について考える．2 行目と，10 行目，14 行目で，分節木をコピーしているが，分節木 $\mathcal{T}$ は，$O(N)$ サイズなので，木全体をコピーするには $O(N)$ 時間かかる．そこで，分節木のコピーをせずに，13 行目でブロック p_1 と p_2 を連結しないことを決定したときに，ブロック p_1p_2 を $\mathcal{T}$ から削除して分節木を前の状態に戻すようにする．分節木からのブロックの削除は $O(1)$ 時間で行えるため，計算時間の削減が期待できる．また，今回のアルゴリズムでは，初期木としてルートノードと各文字でラベル付けされた子のみからなる木を使用したが，接尾辞木の深さ 1 までの部分木を代用することで，訓練に必要なテキスト走査回数を減らせることが期待できる．また，$f_{\mathrm{MDL}}(S, \mathcal{T})$ の計算も，実際にはブロック単位でテキストを走査すれば良いので，$O(M)$ 時間に短縮することができる．

5. おわりに

本稿では，MDL 原理に基づいて分節木に含める文字列を貪欲に決定していく訓練アルゴリズムを提案した．このアルゴリズムは，分節木を再構築していく際に，符号化されたブロック系列の長さの総和と分節木を符号化したものとの和が短くなるかどうかを基準に，登録する文字列の有用性を判断する．また，この基準で自動的に符号語長を調節するため，あらかじめ設定するパラメータが不要である．さらに，提案アルゴリズムは，入力テキスト長を N とすると，$O(N)$ 領域を使って $O(rN^2)$ 時間で動作することを示した．ここで r は訓練中のテキスト操作の回数であり，入力テキストを初期木で分割したときの分割数を M とすると $O(M)$ で抑えられる数である．このように理論的には多項式時間であるものの，現時点の我々の実装では，数メガバイト以上のテキストに対して適用が困難なほど，圧縮に時間がかかることを予備的な実験により確認している．そのため，今回，既存の手法との比較実験を行うことができなかった．よって，実際の圧縮速度が実用的なレベルを達成できるようにアルゴリズムを改善し，効率良い実装を実現することは今後の第一の課題である．

謝　辞

本研究は，文部科学省ブローバル COE プログラム「知の創出を支える次世代 IT 基盤拠点」および科研費（23700002）の助成を受けたものである．

参考文献

1) Abrahams, J.: Code and parse trees for lossless source encoding, *Compression and Complexity of Sequences 1997*, pp.145–171 (1997).
2) Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V. and Rao, S.S.: Representing Trees of Higher Degree, *Algorithmica*, Vol.43, pp.275–292 (2005).
3) Jansson, J., Sadakane, K. and Sung, W.-K.: Ultra-succinct representation of ordered trees, *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics, pp.575–584 (2007).
4) Kida, T.: Suffix Tree Based VF-Coding for Compressed Pattern Matching, *Proceedings of the Data Compression Conference*, p.449 (2009).
5) Klein, S.T. and Shapira, D.: Improved Variable-to-Fixed Length Codes, *Proceedings of the 15th International Symposium on String Processing and Information Re-*

Vol.2011-DBS-152 No.14
Vol.2011-IFAT-103 No.14
2011/8/3

trieval, Lecture Notes in Computer Science, Vol.5280, Berlin, Heidelberg, Springer-Verlag, pp.39–50 (2009).
6) Salomon, D.: *Data Compression: The Complete Reference*, Springer, 4th edition (2006).
7) Salomon, D.: *Variable-length Codes for Data Compression*, Springer (2007).
8) Sayood, K.(ed.): *Lossless Compression Handbook*, Academic Press (2002).
9) Tjalkens, T.J. and Willems, F. M.J.: Variable to Fixed-Length Codes for Markov Sources, *IEEE Transactions on Information Theory*, Vol.IT-33, No.2, pp.246–257 (1987).
10) Tunstall, B.P.: Synthesis of noiseless compression codes, PhD Thesis, Georgia Institute of Technology, Atlanta, GA (1967).
11) Uemura, T., Yoshida, S., Kida, T., Asai, T. and Okamoto, S.: Training parse trees for efficient VF coding, *Proceedings of the 17th international conference on String processing and information retrieval*, Lecture Notes in Computer Science, Vol.6393, Berlin, Heidelberg, Springer-Verlag, pp.179–184 (2010).
12) Visweswariah, K., Kulkarni, S.R., Member, S. and Verdu, S.: Universal Variable-to-Fixed Length Source Codes, *IEEE Transactions on Information Theory*, Vol.47, pp.1461–1472 (2001).
13) Yamamoto, H. and Yokoo, H.: Average-Sense Optimality and Competitive Optimality for Almost Instantaneous VF Codes, *IEEE Transactions on Information Theory*, Vol.47, No.6, pp.2174–2184 (2001).
14) Yoshida, S. and Kida, T.: An Efficient Algorithm for Almost Instantaneous VF Code Using Multiplexed Parse Tree, *Proceedings of the Data Compression Conference*, IEEE Computer Society, pp.219–228 (2010).
15) Ziv, J.: Variable-to-fixed length codes are better than fixed-to-variable length codes for Markov sources, *IEEE Transactions on Information Theory*, Vol.36, No.4, pp. 861–863 (1990).
16) 喜田拓也：STVF 符号：頻度刈り込み接尾辞木を用いた効率良い VF 符号化，*DBSJ Journal*, Vol.8, No.1, pp.125–130 (2009).