Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

VF 符号と算術符号の組合せ手法による圧縮性能の向上について

吉田 諭史[†1]　喜田 拓也[†1]

本稿では，VF 符号に算術符号を組み合わせることで，検索の効率と圧縮率とを保つ方法について議論する．ここで議論する VF 符号とは，分節木と呼ばれる圧縮のための辞書木を用いて元の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データ圧縮を達成する情報源符号化方法である．VF 符号は，近年，パターン照合を高速化することのできるデータ圧縮法として見直されている．VF 符号は，符号語長が固定であるという制限から，分節木が小さいときには圧縮性能が低い．圧縮率を向上させるには分節木を大きくすればよいが，逆にパターン照合時の前処理に時間がかかり全体の検索の速度を低下させてしまう．そこで，VF 符号の出力を，展開が早い他の符号化方法で符号化することで，圧縮率と検索速度の両方を保つ方法が考えられる．本稿では，代表的な VF 符号である Tunstall 符号および VF 符号の中では優れた圧縮性能を持つ STVF 符号に Range Coder を組み合わせた圧縮方法について，その圧縮性能を実験的に評価した．その結果，符号語長が短い場合において，それぞれおよそ 18 ∼ 20%，7 ∼ 15% の圧縮率改善が見られた．

A Combination of VF Coding with Arithmetic Coding for Efficient Compression

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

In this paper, we discuss about a method to preserve both search efficiency and compression ratio by combining VF coding with arithmetic coding. A VF code, we discuss here, is a source coding scheme that parses an input text into variable-length blocks with a dictionary tree, called parse tree, and then encodes them by fixed-length codewords. It is reevaluated recently since it can accelerate pattern matching on the text. As the codewords are fixed length, a VF code usually has low compression ratios when its parse tree is small. Although we can obtain higher compression ratios when we make the parse tree large enough, pattern matching speed will decline since we take much more time to preprocess such large tree for compressed pattern matching. One of the natural trade-off solutions is to encode outputs of a VF code with the other compression method whose decompression speed is fast. We would be able to do fast pattern matching on such compressed text by decoding it to a sequence of VF codewords once and then searching on the sequence without decoding the codewords. In this paper, we investigated a combination of Tunstall codes or STVF codes with Range coder. The experimental results show that those combinations can improve the compression ratios about 18 ∼ 20% and 7 ∼ 15%, respectively, when the codeword length is short.

1. はじめに

圧縮された文書に対して高速に情報検索を行うという要求から，圧縮されたテキストデータに対して，それを明示的に復元することなくパターン照合を行う圧縮パターン照合 (Compressed pattern matching) と呼ばれる問題が生じた．この問題は 90 年代初頭[1)] で提案され，以降盛んに研究が行われている．当初は，主に理論的な興味から始まったが，大規模テキストデータベースが個人で比較的簡単に取り扱えるようになった今日では，実用上の重要な課題となってきている．

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[8),12)]．情報源符号化手法を，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 写像として見ると，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，大抵の LZ 系圧縮法は VV 符号とみることができる．

†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.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

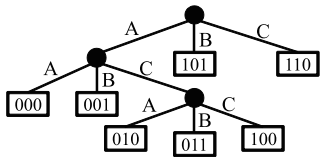

図 1 分節木の例

これまで，多くの場合において，テキスト圧縮で重要視されてきた要素は圧縮率である．そのため，現在では，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[9]．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，固定長符号である VF 符号が見直されている[3),5)]．

代表的な VF 符号である Tunstall 符号[13)] は，分節木と呼ばれる辞書木を用いて入力テキストを分割する（図 1）．分節木に登録されている語にはそれぞれ固定長の符号語が割り当てられ，分割したテキストの部分文字列（ブロックと呼ばれる）を順に対応する符号語の列へと置きかえることで符号化を行う．Tunstall 符号は Huffman 符号と同様に，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[17)]．

VF 符号の欠点である圧縮率の低さを改善するため，Klein，Shapira ら[5)] と Kida[3)] 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]．Tunstall 符号では，入力テキストとして記憶のない情報源を仮定していた．これに対して，接尾辞木を用いたこれらの手法では，対象となるテキストが事前に与えられていることを仮定している（すなわち，オフラインなデータ圧縮法である）．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[17)]．

VF 符号の圧縮率は，分節木の質と大きさによってきまる．分節木の質に関しては，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 完全以上であると目されている[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するだけ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符号語系列は短くなる．ただし，復号時にも分節木の情報が必要であるため，かえって圧縮データ全体としてはサイズが大きくなってしまう．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．

圧縮パターン照合の高速化という観点からも，符号語長を長くして分節木を大きくしすぎると，パターン照合のための前処理に多くの時間がかかるという問題が指摘されている[17)]．したがって，高い圧縮率を得られる小さな分節木を構築することが求められるが，先に述べたように分節木の質を高めることは難しい．

ここでの問題は，短い符号語を用いた VF 符号の圧縮率をいかにして向上させるか，ということである．分節木による VF 符号の出力を，比較的展開速度が速く圧縮率の良い他の符号化で符号化する手法は単純だが有効だと考えられる．便宜的に，元テキストに対する VF 符号を**内側**の符号と呼び，その出力に対する符号を**外側**の符号と呼ぶことにする．すなわち，この多段の符号化テキストに対するパターン照合は，外側の符号を展開しながら内側の符号語に対しては直接的に行うことで実現される．本稿では，内側の符号として Kida が提案した STVF 符号[3)] および Tunstall 符号[13)] を，外側の符号化手法として Range Coder[6)] を選択し，その圧縮性能を調査した．その結果，符号語長が短い場合（$\ell = 8 \sim 12$）において，Tunstall 符号と Range Coder の組み合わせでは約 18 ∼ 20%，STVF 符号と Range Coder の組み合わせでは約 7 ∼ 15% の圧縮率改善が見られた．

2. 準　　備

Σ を**有限アルファベット**とする．Σ の要素を**文字**と呼ぶ．自然数 i に対して，Σ^i を Σ の要素を i 個並べた列の集合，つまり，$\{x_1x_2 \ldots x_i \mid x_j \in \Sigma\}$ とする．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 $|\Sigma|$ とかける．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**と呼び，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**と呼ぶ．

任意の文字列 $x \in \Sigma^*$ のテキスト T 中の**出現確率**を $\Pr_T(x)$ とかく．また，便宜上 $\Pr_T(\varepsilon) = 1$ と定義する．当然 $\Pr_T(x)$ は T に依存するが，文脈から対象のテキストが

⋆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[3)] 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.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

明らかな場合や，情報源の統計的性質として取り扱う場合には，単に $\Pr(x)$ とかく．

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**）と呼ぶ．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**と呼ぶ．根をもつ木を**根つき木**と呼ぶ．さらに，k 分木において，子の数が k である内部ノードを**完全内部ノード**と呼ぶ．すべての内部ノードが完全内部ノードであるものを，**完全 k 分木**と呼ぶ．

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 $T \in \Sigma = \{a_1, a_2, \ldots, a_k\}$ 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**と呼ぶ．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．

今，テキストが記憶のない情報源からの系列であると仮定しよう．この場合には，**Tunstall 符号**[13] が最適な符号化を与える（文献[10]）．Tunstall 符号は，以下のようにして構築される **Tunstall 木**と呼ばれる完全 k 分木を用いる．

情報源記号 $a_i \in \Sigma$ の出現確率を $\Pr(a_i)$ とする．このとき，分節木の根から節点 μ へ至るパスを辿る文字列 $x_\mu \in \Sigma^+$ の出現確率は $\Pr(x_\mu) = \prod_{\eta \in \xi} \Pr(\eta)$ である．ここで，ξ は根から μ までのパス上のラベル列である．ブロックの平均長を最大にするという意味で，最適な分節木 $\mathcal{T}^*$ は次のようにして構築できる．

(1) $\mathcal{T}^*$ を，根と $k = |\Sigma|$ 個の子からなる深さ 1 の順序付き k 分木とする．これを初期木 $\mathcal{T}_0$ と呼ぶ．
(2) 以下のステップを，$\mathcal{T}^*$ の葉の個数 L が 2^ℓ を超えない間，繰り返す．
  (a) $\mathcal{T}^*$ の葉のうち，最大の確率を持つ葉 v を選択する．
  (b) 初期木 $\mathcal{T}_0$ を v に接ぎ木する．つまり，v の下に k 個の子供を追加し，v を内部ノード化する．このとき，葉の数 L は $k-1$ 個分だけ増える．

木 $\mathcal{T}^*$ の内部ノードの個数を m とすると，$\mathcal{T}^*$ の葉の総数は $L = m(k-1)+1$ なので，符号語長 ℓ は条件 $m(k-1)+1 \leq 2^\ell$ を満たす必要がある．したがって，テキストを符号語長 ℓ で符号化するならば，内部ノードが $m = \lfloor (2^\ell - 1)/(k-1) \rfloor$ となる大きさの分節木を構築することになる．

分節木の大きさは，パターン照合時の前処理時間に影響を与える．分節木が大きければ大きいほど圧縮率が高くなる傾向にあるが，逆にパターン照合のための情報を分節木から計算するための時間が余計にかかってしまう．一方，パターン照合におけるテキスト走査の部分は，符号系列の長さに比例するので，圧縮率の高いほうが有利に働く．当然，パターン照合時間は用いるパターン照合アルゴリズムにも依存する．理論的には，VF 符号は**正規 collage system**[4] と呼ばれる族に属するので，VF 符号上で動作可能な Aho-Corasick 型や Boyer-Moore 型のパターン照合アルゴリズムを組織的に導出することができる．

4. STVF 符号

STVF 符号では，入力テキストに対する接尾辞木を構築した後，各ノードの頻度を元に適切に刈り込むことで分節木を構築する．まずは，分節木の土台となる接尾辞木について簡単に説明する．

接尾辞木 (suffix tree) は，文字列の全ての接尾辞を格納する根つき木である．図 2 は，テキスト T = BABCABABBABCBAC$ に対する接尾辞木の例である．形式的には，テキスト T の接尾辞木 $ST(T)$ を次のように定義する：

(1) ルートを除くすべての内部ノードは，少なくとも 2 つの子をもつ，
(2) すべての枝は空語ではない文字列でラベルづけされる，
(3) すべての内部ノード u について，u の子はそれぞれ異なる文字で始まるラベルをもつ，
(4) 接尾辞木 $ST(T)$ のノード u について，$str(u)$ を $ST(T)$ のルートから u までのパス上にある辺のラベルを順に連結した文字列とする．このとき，T の任意の部分文字列

Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

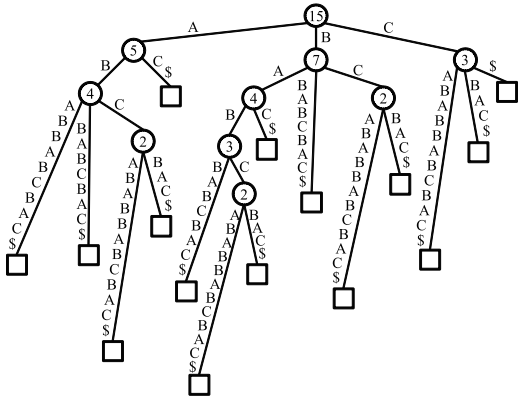

図 2 T = BABCABABBABCBAC$ に対する接尾辞木.
四角は葉を，黒丸は内部ノードをそれぞれ示している．各葉は T の一つの接尾辞に対応する．円の中の数字はそのノードの頻度を示す.

x について，x は $str(u)$ の接頭辞となる $ST(T)$ のノード u が存在する．

また，ノード u の出現頻度を T 中に $str(u)$ が出現する回数と定義し，これを $f(u)$ とかく．以降，特に混乱のない限り，u と $str(u)$ を同一視する．接尾辞木のノード数は $O(|T|)$ であり，それに比例した時間で構築することができる[2]．各ノードの出現頻度は，そのノードをルートとする部分木の葉の数となる．したがって，$ST(T)$ を帰りがけ順に探索することにより $O(|T|)$ 時間ですべてのノード（すなわち T の任意の部分文字列）の出現頻度を計算することができる．

接尾辞木 $ST(T)$ は，その最も深い葉がテキスト T 全体を表しているため，接尾辞木全体を分節木として用いることはできない．そこで，STVF 符号は，文字列 T の接尾辞木を適切に短く刈り込んだものを分節木として用いる．接尾辞木 $ST(T)$ を刈り込んでできた木を**刈り込み接尾辞木**とよぶ．また，刈り込んで，葉の個数が L 個となったものを $ST_L(T)$ とかく．元の接尾辞木 $ST(T)$ において深さが 1 であるノードをすべて持つような刈り込み接尾辞木は，T 中のすべての記号を含んでいることに注意する．

いま，符号語長 ℓ でテキスト T を符号化するとしよう．接尾辞木中のどのような文字列に符号語を割り当てるかによって圧縮率は変化する．符号語長は固定なので，符号系列を短くするには，$L \leq 2^\ell$ の条件のもとで入力テキストの分割数を最小にすればよい．ただし，一方で，刈り込み接尾辞木上に残る文字列の長さの総和を短くしなければならない．なぜなら，接尾辞木上の最も長い系列に符号語を割り当てれば入力テキストの分割数は 1 となり最小であるが，圧縮テキストとしてその木自身も保存しておかなくてはならないので，これは無意味である．文献[5] で議論されているように，最適な刈り込みを求めることは困難であるため，現実的には刈り込みアルゴリズムはヒューリスティックなものを使わざるを得ない．

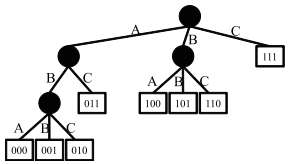

図 3 T = BABCABABBABCBAC に対する STVF 符号のための刈り込み接尾辞木
四角は符号語が割り当てられている葉を示しており，中に符号語が書かれている．それ以外の黒丸は，内部ノードである.

STVF 符号[3] の刈り込み戦略は，ルートとその直下の子からなる木を初期の刈り込み接尾辞木 $ST_k(T)$ とし，$ST(T)$ をルートから幅優先探索しつつ，浅い位置にあるノードのうち頻度の高いものから順にそのノードのすべての子を $ST_L(T)$ $(L \geq k)$ へ加え，葉の数 L が $L \leq 2^\ell$ を満たさなくなったら $ST_L(T)$ の伸長をやめるというものである．ただし，元の接尾辞木 $ST(T)$ において葉を示す子を加えるときは，その葉へ向かうラベルの先頭 1 文字のみを残す．また，符号語は木 $ST_L(T)$ の葉にのみ割り当てられ，前節で述べた Tunstall 符号と同じ符号化の手順で符号化を行う． 図 3 は，テキスト T = BABCABABBABCBAC に対する STVF 符号のための刈り込み接尾辞木である．図 3 の分節木を用いて T を分割すると，BA/BC/ABA/BB/ABC/BA/C となる．すなわち，100/110/000/101/010/100/111 と符号化される．

5. Range Coder

Range Coder[6] は，算術符号[7] の一種である．算術符号は，情報源系列の**累積確率**に対して符号化を行い，系列全体を一つの符号語として表現する．十分に長い情報源系列に対して，Huffman 符号よりも良い圧縮率を得ることができる手法であることが知られている．

いま，記憶のない情報源を仮定する．このとき，空でない文字列 $x = x_1x_2\cdots x_n$ の出現確率は $\Pr(x) = \prod_{i=1}^{n} \Pr(x_i)$ である．また，文字列 x の累積確率を $C_f(x) = \sum_{t\in\{y|y\in\Sigma^{|x|},y\prec x\}} \Pr(y)$ と定義する．ここで，$\prec$ は文字列の辞書順による大小を表す関係とする．すなわち，$y \prec x$ は，文字列 x よりも y のほうが辞書順で前にある文字列であ

Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

アルゴリズム Arithmetic coding
入力 文字列 x
出力 符号化された列

1: $lower \leftarrow 0$
2: $range \leftarrow 1$
3: **while** テキストの最後ではない **do**
4: $c \leftarrow$ テキストの次の文字
5: $lower \leftarrow lower + range \times C_f(c)$
6: $range \leftarrow range \times \Pr(c)$
7: **end while**
8: **return** $lower$

図 4 算術符号化による圧縮

ることを示す．最初に提案された算術符号のアルゴリズム[7] は，図 4 のとおりである．ここで，$lower$ と $range$ は，それぞれ現在まで読み込んだテキストの累積確率と出現確率を表している．

このアルゴリズムによる符号化には 2 つの問題がある．まず，実数を任意の精度で計算できる必要がある．実数の精度に上限がある環境では，入力テキストが長くなったとき，文字列の出現確率が極めて小さくなり，やがて $lower$ や $range$ が表現できなくなる．もうひとつの問題点は，実数値の演算は比較的コストが高いため，圧縮や伸長に多大な時間が必要になることである．

Range Coder では，実数の代わりに固定長の整数を使い，これらの問題を回避している．つまり，図 4 のアルゴリズムにおける $lower$ と $range$ に対応する値を固定長の整数の計算で代用する．各文字の出現確率も，テキスト中の頻度で管理を行い，必要なときにテキストの長さで割ったものをオンザフライで計算して用いる．このように，$lower$ と $range$ の更新を整数の演算のみで実現することで高速な符号化を実現する．このとき，入力テキストを読み込むにつれ，$range$ の値が徐々に小さくなり圧縮が続行できなくなる．そこで，$range$ が事前に決めておいたあるしきい値を下回ったときに，$lower$ と $range$ の双方に定数 M を掛けることでこれを回避する．ただし，$lower$ に整数をかけ続けるので，そのままでは固定

アルゴリズム Range Coder
入力 文字列 x
出力 符号化された列

1: $range \leftarrow initRange$
2: $lower \leftarrow initLower$
3: **while** 入力テキスト x の最後ではない **do**
4: $c \leftarrow$ 入力テキストの次の文字
5: $lower \leftarrow lower + CF(c) \times \lfloor range/|x| \rfloor$
6: $range \leftarrow F(c) \times \lfloor range/|x| \rfloor$
7: **while** $range < minRange$ **do**
8: $lower$ の最上位部分を出力する
9: $range \leftarrow range \times M$
10: $lower \leftarrow lower \times M$
11: **end while**
12: **end while**
13: $lower$ を出力する

図 5 Range Coder による圧縮

長の整数で計算できないため，問題がある．この問題を解消するために，$lower$ に整数をかける際に，$lower$ の最上位の部分を出力する．この解決策は，繰り上がりによって出力した部分に変更が生じるため問題があるようにみえる．しかしながら，$lower$ は最終的には $[lower, lower + range]$ に収まり，$lower$ と $lower + range$ の最上位の共通部分は，繰り上がりによって桁が変化しない部分である．したがって，この上位部分を出力しても問題が発生しない．

Range Coder による符号化アルゴリズムを図 5 に示す．ここで，$initRange$, $initLower$, M, $F(c)$ と $CF(c)$ はそれぞれ $range$ の初期値，$lower$ の初期値，$lower$ にかける整数の値，文字 c の出現頻度，累積出現頻度である．なお，文字 a の累積出現頻度を $\sum_{b \prec a} F(b)$ と定義する．

Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

6. VF 符号に対するエントロピー符号化

先に述べたとおり，VF 符号の出力は固定長のビット列で表現された整数の列であり，一つ一つが元テキストの可変長の部分文字列に対応している．分節木を大きくすれば（符号語長 ℓ を大きくすれば）するほどブロックの平均長は長くなるが，辞書として記憶すべき分節木の保存コストが増大する．また同時に，各符号語の出現頻度が全体的に低くなり，有限の長さのテキストを圧縮する場合では使用されない無駄な符号語の数が増大する．

したがって，VF 符号と FV 符号を組み合わせて VV 符号を構成することは自然な考えである．Savari と Szpankowski らは，Tunstall 符号と，Huffman 符号および Shannon-Fano 符号を組み合わせた方式について解析し，次のような結果を導いている[11]．

定理 1 (Savari と Szpankowski ら[11] の定理 1) $\mathcal{H}$ は無記憶 2 元情報源のエントロピーとし，$R_{\text{T-SF}}$ および $R_{\text{T-H}}$ を，それぞれ Tunstall-Shannon-Fano 符号と Tunstall-Huffman 符号の冗長度とすると，以下の式が成り立つ．

$$\limsup_{M\to\infty} \log_2 M \cdot R_{\text{T-SF}}(M) < \mathcal{H},$$
$$\limsup_{M\to\infty} \log_2 M \cdot R_{\text{T-H}}(M) < \mathcal{H}\log_2(2e^{-1}\log_2 e).$$

また，我々は，実際に Tunstall-Huffman 符号を実装し，その性能について実験的に評価を行い，Tunstall 符号や Huffman 符号を単体で用いるよりも大幅に圧縮率を向上できることを示している[16]．

本稿では，内側の符号化として STVF 符号あるいは Tunstall 符号を用い，外側の符号化として Range Coder を組み合わせる．VF 符号によって出力される各符号語は，範囲 $[0..2^{\ell}-1]$ の整数である．Range Coder は，理論上は情報源アルファベットのサイズによらない手法であるが，ウェブ上から入手可能な既存プログラムは入力として ASCII コードもしくはバイトコードを仮定しているものが大半であり，2^8 を超える情報源アルファベットに対応しているものを入手できなかった．そのため，今回，実用的な範囲の符号長（ℓ が 7〜20 程度）の VF 符号の出力を符号化できる Range Coder を独自に実装している．

7. 実　　験

Tunstall 符号と STVF 符号，Range Coder を実装し，圧縮時間と展開時間，圧縮率の比較を行った．プログラムはすべて C++言語で実装し，GNU g++3.4 でコンパイルした．比較した手法は，Tunstall 符号と STVF 符号，Tunstall 符号の外側に Range Coder を施したもの (Tunstall-RangeCoder)，STVF 符号の外側に Range Coder を施したもの (STVF-RangeCoder) の 4 つである．また，符号語長は，8 ビットから 16 ビットの 1 ビット刻みとした．テキストとして，The Canterbury Corpus のうち bible.txt(英文，4MB) を使用した．なお，実験環境は表 1 のとおりである．

表 1　実験環境

CPU	Intel(R) Xeon(R) プロセッサ 3.00GHz デュアルコア，Hyper Threading 対応
メモリ	12GB
OS	Red Hat Enterprise Linux ES Release 4

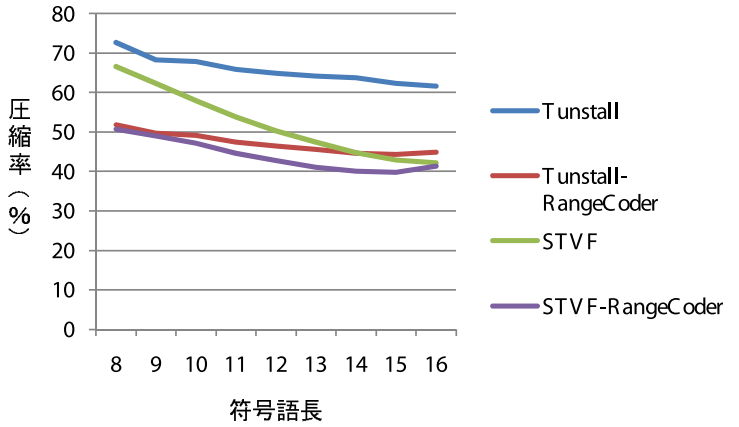

図 6　圧縮率の比較

圧縮率の比較を図 7 に示す．Tunstall 符号と STVF 符号のどちらも外側に Range Coder を施したほうが圧縮率が向上することがわかる．圧縮率の向上度は Tunstall 符号と Range Coder の組み合わせでは約 18 ～ 20%，STVF 符号と Range Coder の組み合わせでは約 7 ～ 15% である．

次に，圧縮時間の比較を図 7 に示す．圧縮時間は，10 回の平均を測定したものである．Tunstall 符号と STVF 符号のどちらも外側に Range Coder を施しても，ほとんど変わらないばかりか，若干ではあるが，速くなっている．これは，圧縮率の向上に伴い，入出力するデータ量が削減されたからであると考えられる．

Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

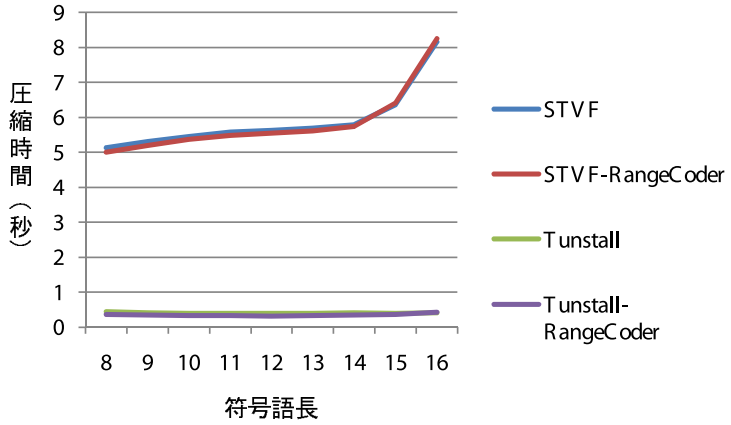

図 7 圧縮時間の比較

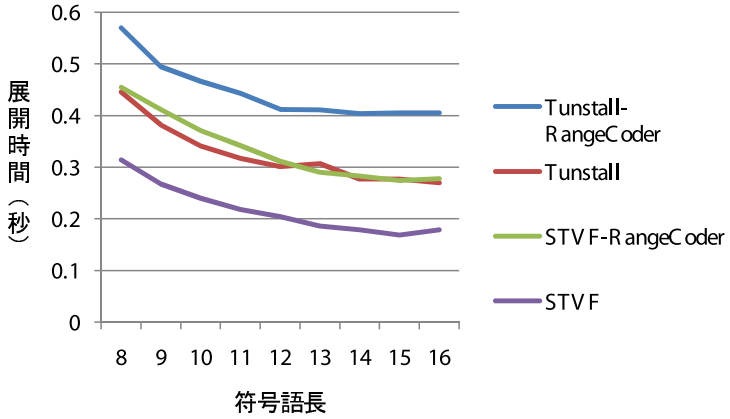

図 8 展開時間の比較

最後に展開時間の比較を図 8 に示す．展開時間も圧縮時間と同様に，10 回の平均を測定したものである．Tunstall 符号と STVF 符号のどちらも外側に Range Coder を施したものは，展開に 0.1~0.14 秒余分にかかっている．これは，Range Coder の展開にかかる時間である．

8. おわりに

本稿では，Tunstall 符号および STVF 符号に Range Coder を組み合わせた場合の圧縮性能について実験的に評価した．その結果，符号語長が短い場合（$\ell = 8 \sim 12$）において，Tunstall 符号と Range Coder の組み合わせでは約 18 ~ 20%，STVF 符号と Range Coder の組み合わせでは約 7 ~ 15% の圧縮率改善が見られた．Range Coder との組み合わせによる圧縮速度の付加コストは見られないので，圧縮率の改善という点では組み合わせることに意味がある．一方，展開速度については，無視できないコストがかかっており，圧縮検索時での影響は少なくないと予想される．本稿では，圧縮性能に着目して議論を行ったが，圧縮パターン照合の時間についての比較実験が今後の重要な課題である．Range Coder よりも展開速度の速い符号化の検討も今後の課題である．

参 考 文 献

1) Amir, A. and Benson, G.: Efficient two-dimensional compressed matching, *Proc. DCC'92*, pp.279–288 (1992).
2) Giegerich, R. and Kurtz, S.: From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction, *Algorithmica*, Vol.19, No.3, pp. 331–353 (1997).
3) Kida, T.: Suffix Tree Based VF-Coding for Compressed Pattern Matching, *Proc. of Data Compression Conference 2009(DCC2009)*, p.449 (2009).
4) Kida, T., Matsumoto, T., Shibata, Y., Takeda, M., Shinohara, A. and Arikawa, S.: Collage system: a unifying framework for compressed pattern matching, *Theor. Comput. Sci.*, Vol.298, No.1, pp.253–272 (2003).
5) Klein, S.T. and Shapira, D.: Improved Variable-to-Fixed Length Codes, *SPIRE '08: Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, Berlin, Heidelberg, Springer-Verlag, pp.39–50 (2009).
6) Martin, G. N.N.: Range Encoding: An Algorithm for Removing Redundancy from a Digitised message, *Proc. Video and Data Recording Conference*, pp.24–27 (1979).
7) Rissanen, J. and Langdon, G.: Arithmetic coding, *IBM Journal of Research and Development*, Vol.23, No.2, pp.149–162 (1979).
8) Salomon, D.: *Data Compression: The Complete Reference*, Springer, 4th edition

Vol.2010-DBS-150 No.10
Vol.2010-IFAT-99 No.10
2010/8/4

(2006).
9) Salomon, D.: *Variable-length Codes for Data Compression*, Springer (2007).
10) Savari, S.A.: Variable-to-Fixed Length Codes for Predictable Sources, *In Proc. of DCC98*, pp.481–490 (1998).
11) Savari, S.A. and Szpankowski, W.: On the Analysis of Variable-to-Variable Length Codes, *In Proc. of ISIT2002*, p.176 (2002).
12) Sayood, K.(ed.): *Lossless Compression Handbook*, Academic Press (2002).
13) Tunstall, B.P.: Synthesis of noiseless compression codes, PhD Thesis, Georgia Inst. Technol., Atlanta, GA (1967).
14) Uemura, T., Yoshida, S. and Kida, T.: An Improvement of STVF Code by Almost Instantaneous Encoding, Technical report, Hokkaido University, Division of Computer Science (2010).
15) Yamamoto, H. and Yokoo, H.: Average-Sense Optimality and Competitive Optimality for Almost Instantaneous VF Codes, *IEEE Trans. on Information Theory*, Vol.47, No.6, pp.2174–2184 (2001).
16) 松井雄大，喜田拓也：Tunstall-Huffman 符号化の効率について，技術報告 i1-28，DEIM Forum 2008, DE 研究会，松山 (2009 年). インタラクティブセッション.
17) 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, p. (2009).