信州大学工学部

学士論文

算術符号化を用いた
無限深さ文脈木重み付け法の実現

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　09T2805K
氏名　佐藤 光

2014 年 2 月 21 日

目次

1 序章

1.1 研究の背景と目的

現在の社会で欠かすことのできないインターネットをはじめとする情報通信システムにおいて扱うデータ量は莫大になってきており，データを保存する為に必要な記憶容量やデータ転送におけるコストを削減することが出来るデータ圧縮は必要不可欠なものである．データ圧縮方法には様々なものがあるが, 本研究ではそのデータ圧縮方法の中でも文脈木重み付け法 (Context-Tree Weighting, CTW) というものについて考える．CTW 法は情報源の確率推定と算術符号に基づく方法であり，基本的な分類として Willems, Shtarkov and Tjalkens によって提案された有限深さ文脈木重み付け法 [1]，Willems によって提案された無限深さ文脈木重み付け法 [2] の 2 種類がある．この CTW 法は，モデルとパラメータが未知であるという条件のもとで，木情報源を効率良く最適に符号化する確率推定法である．一般に確率モデルに基づくデータ圧縮では，情報源がどのような確率構造になっているかを調べ，確率に応じて符号化を行う必要がある．情報源の確率構造は，符号化に先立ち情報源系列全体を調べることによって得ることができる．一方，本研究で取り上げる CTW 法では，情報源の確率構造を予め把握せずに符号化を行いながら情報源の確率構造を調べることが出来る．これは，情報を受信する際などに全ての情報が揃う前に復号化を開始できるという利点がある．

有限深さ文脈木重み付け法は出現しない文脈が多くメモリの使用効率が悪いため，三澤により効率よくメモリを使用する方法 [3] が提案されている．一方，無限深さ文脈木重み付け法は系列に従って文脈木が成長していくために, 大きなファイルほど多くのメモリを必要とする. そこで, 生成できるセグメント数に制限を付けることで, プログラムを有限メモリで動作させる方法 [4] が三澤により提案された．普通 CTW 法では, 算術符号化を用いて圧縮率の算出を行うが，[4] における検証ではこれを行っておらず，符号化確率から符号語長を近似計算している．本研究では実際の算術符号化による圧縮率の算出を行った．その結果，2 つの方法で圧縮率に変化が無いことを確認できた．また，セグメントの更新頻度に対する考察を行い，不要なセグメントの存在が示唆された．

1.2 本論文の構成

本論文では，次のような構成を取る．2 章では木情報源，CTW 法と有限深さ文脈木重み付け法のアルゴリズムついて説明する．3 章では無限深さ文脈木重み付け法と，セグメントの概念について説明する．4 章では [4] の方法と算術符号を用いた方法の比較を行う．5 章ではセグメントの更新頻度に関する考察を行う．6 章でまとめをする．

2　文脈木重み付け法

CTW 法は未知の木情報源から出力される系列の確率を推定するために，各モデルに対して行ったパラメータ推定をモデル全体の重み付けで混合する．つまりエンコーダとデコーダは真のモデルとパラメータを知らない状況で符号化を開始する．

本章では有限深さ文脈木重み付け法を元に文脈木重み付け法について説明し，続く 3 章で，有限深さの拡張として無限深さ文脈木重み付け法について説明する．

2.1　木情報源

はじめに，本論文では x_m^n で系列 x_m，x_{m+1}，…，x_n を表し，$n < m$ ならば $x_m^n = \lambda$ と定義する．λ は空系列を意味している．木情報源とは系列 $x_{-\infty}^{\infty}$ を生成する Markov 情報源の一種である [1]．アルファベットを $\{0,1\}$ と仮定する．また，文字列の語尾を接尾辞 (suffix) と言う．具体的には x_m^n の接尾辞は λ，x_n^n，x_{n-1}^n，…，x_{m+1}^n，x_m^n である．ある文字列の集合を考え，どの文字列も他の文字列の接尾辞になっていない場合その文字列の集合は接尾辞フリー (suffix-free) であると言う．$|s|$ を文字列 s の長さを表す演算子とする．

完全な接尾辞フリーな有限集合 $\mathcal{S}$ を考える．接尾辞フリーな集合 $\mathcal{S}$ が完全であるとは $\sum_{s\in\mathcal{S}} 2^{-|s|} = 1$ が成り立つことである．この時，$\mathcal{S}$ に対して完全木が対応し，各 s はその完全木の葉に対応する．例として完全な接尾辞フリーな集合 $\mathcal{S} = \{00, 10, 1\}$ に対して図 1 のような完全木が対応する．

長さ D 以下の文字列からなる完全接尾辞フリーな集合 $\mathcal{S}$ を考える．各文字列 $s \in \mathcal{S}$ にパラメータ θ_s を対応させる．各パラメータ θ_s は $[0,1]$ の値を取り $\{0,1\}$ 上の分布を表す．パラメータ全体をパラメータベクトル $\Theta_s \stackrel{\text{def}}{=} \{\theta_s : s \in \mathcal{S}\}$ と表す．このとき，次のような確率構造でシンボルを出力する情報源を考える．今までに x_m^n が出現している時 D シンボル遡って系列 x_{n-D+1}，x_{n-D+2}，…，x_n を見れば，この系列の接尾辞である $\mathcal{S}$ の要素 s は一意に決まる．この時 s に対応する θ_s が決まり，θ_s が表す分布に従って x_{n+1} の値が確率的に決まる．このように定義される Markov 情報源を木情報源と呼ぶ [1]．また，このとき $\mathcal{S}$ を木情報源のモデルと呼び，$\mathcal{S}$ に属する文字列を文脈と呼ぶ．

2.2　有限深さ文脈木重み付け法

深さ D の文脈木重み付け法とは，長さが D 以下の文字列からなるモデルを持つ木情報源に対し，モデル，パラメータ両方が未知という条件で次のシンボルの出現確率を推定するための方法である．この推定法は，シンボルあたりの個別冗長度を漸近的に 0 にするという意味で最適な方法である．有限深さ文脈木重み付け法の最適性は Willems, Shtarkov and Tjalkens

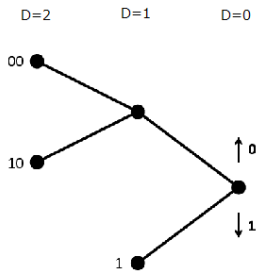

図 1　完全木の例

によって証明されている [1]．2.3 節ではパラメータの推定，すなわち各文脈におけるシンボルの発生確率を推定する方法を説明する．2.4 節ではモデルのコストの概念を導入する．モデルのコストは，長さが D 以下の文字列からなるすべてのモデルの上に確率分布を定義する．2.5 節では各モデルを用いたときの推定値を，モデルのコストに基づいた重みにしたがって混合するための便利な漸化式を導く．このように混合された推定値が文脈木重み付け法の推定値である．

2.3　推定器

CTW 法ではパラメータ θ_s を推定するための推定器は任意のもので良い．本研究では [1] に従い 2 値の情報源アルファベットの確率分布を推定する方法である Krichevsky-Trofimov(KT) 推定を用いる．今，0,1 が出現した回数をそれぞれ a, b とすると 0 が a 個，1 が b 個含まれる情報源系列の出現確率の推定値 $P_e(a,b)$ は以下のように定義される．

$$P_e(a,b) \stackrel{\text{def}}{=} \int_0^1 \frac{1}{\pi\sqrt{(1-\theta)\theta}}(1-\theta)^a\theta^b d\theta \tag{1}$$

P_e は次のような逐次的な関係を持つ．

$$P_e(0,0) = 1 \tag{2}$$

$$P_e(a+1,b) = \frac{a+\frac{1}{2}}{a+b+1} \cdot P_e(a,b) \tag{3}$$

$$P_e(a,b+1) = \frac{b+\frac{1}{2}}{a+b+1} \cdot P_e(a,b) \tag{4}$$

従って，式 (1) を (2)(3)(4) のように漸化式で表すことにより四則演算のみで逐次計算することができる．

2.4 モデルのコスト

モデル $\mathcal{S}$ を用いて符号 code(s) を以下のように定義する．

$$\mathrm{code}(s) = \begin{cases} \lambda & (|s| = D) \\ 0 & (s \in \mathcal{S} \text{ かつ } |s| < D) \\ 1\mathrm{code}(0s)\mathrm{code}(1s) & (s \notin \mathcal{S} \text{ かつ } |s| < D) \end{cases} \tag{5}$$

この時，code(λ) が表す符号語の長さを $\mathcal{S}$ のコストと呼び次の式で表せる．

$$\Gamma_D(\mathcal{S}) = |\mathcal{S}| - 1 + |\{s : s \in \mathcal{S}, |s| \neq D\}| \tag{6}$$

ここで，$D = 2$ とした場合について考える．考えうるモデルには図 2 のような 5 種類が存在する．

図 2(a) のモデルについて考える．図 2(a) は根だけからなるモデルになるが，このモデルに対しては式 (5) の 2 つ目の条件式より，code(λ)=0 となることがわかる．ここで，code(λ) が表す符号語の長さがモデルのコストとなるため，このモデルのコストは 1 ということになる．

一方，図 2(e) のモデルについて考えると，根 λ に対しては $\lambda \notin \mathcal{S}$ かつ $|\lambda| < D$ が成り立っていることがわかる．すなわち式 (5) の 3 つ目の条件式より,code(λ)=1 code(0) code(1) となる．code(0), code(1) はそれぞれ根の子ノードに対応しているため, ここでも ($s \notin \mathcal{S}$ かつ $|s| < D$) の条件が成り立つ．よって, code(0)=1 code(00) code(10), code(1)=1 code(01) code(11) となる．ここで code(00)，code(10)，code(01)，code(11) については ($|s| = D$) の条件が成り

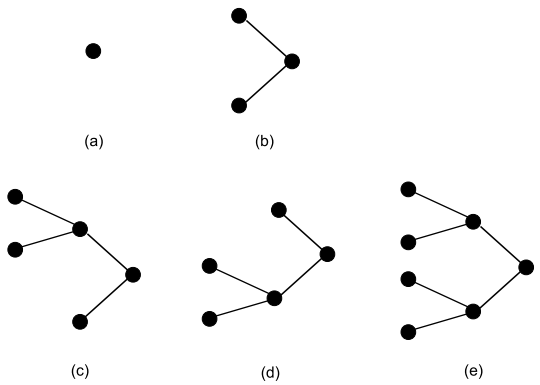

図 2　モデルの種類

立つため，code(s)=λ となることから消える．以上のことをまとめると式 (7) のようになる．

$$\begin{aligned}\mathrm{code}(\lambda) &= 1\ \mathrm{code}(0)\ \mathrm{code}(1)\\ &= 1\ 1\mathrm{code}(00)\mathrm{code}(10)\ 1\mathrm{code}(01)\mathrm{code}(11)\\ &= 1\ 1\ 1\end{aligned} \tag{7}$$

すなわち，このモデルのコストは 3 となることが分かる．(b)，(c)，(d) 図のコストについても同様にして求めると，全て 3 となる．

2.5 重み付け確率

全ての葉が深さが D である完全木を文脈木と呼び，各ノードは文脈に対応する．また，x_{t-D}^{t-1} は葉に対応する．ここで，文脈 $s = x_{t-d}^{t-1}$ に対応するノードの深さを $d(d \leq D)$ とし，深さ $D-d$ 以下のモデル全体を $\mathcal{C}_{D-d}$ とする．$a_s(x_1^t)$，$b_s(x_1^t)$ をそれぞれ系列 x_1^t が出現した際の文脈 s の次に 0，1 が出現した回数として

$$P_e^s(x_1^t) \stackrel{\mathrm{def}}{=} P_e(a_s(x_1^t), b_s(x_1^t)) \tag{8}$$

$$P_e^s(x_t|x_1^{t-1}) \stackrel{\mathrm{def}}{=} \begin{cases} \dfrac{a_s(x_1^{t-1})+\frac{1}{2}}{a_s(x_1^{t-1})+b_s(x_1^{t-1})+1} & (x_t=0)\\ \dfrac{b_s(x_1^{t-1})+\frac{1}{2}}{a_s(x_1^{t-1})+b_s(x_1^{t-1})+1} & (x_t=1)\end{cases} \tag{9}$$

と定めると，式 (3)(4) より次の式が成り立つ．

$$P_e^s(x_1^t) = P_e^s(x_t|x_1^{t-1}) \cdot P_e^s(x_1^{t-1}) \tag{10}$$

また，式 (6) のモデルのコストを用いると文脈 s の重み付け確率 $P_w^s(x_1^t)$ は次の式で定義される．

$$P_w^s(x_1^t) \stackrel{\mathrm{def}}{=} \sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} \prod_{u\in\mathcal{U}} P_e^{us}(x_1^t) \tag{11}$$

$$\text{このとき} \sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} = 1 \tag{12}$$

式 (12) はいわゆる等号が成立しているクラフトの不等式 [6] である．よって $2^{-\Gamma_{D-d}(\mathcal{U})}$ は深さが $D-d$ 以下のモデル $\mathcal{U}$ への重みを意味している．また，$\prod_{u\in\mathcal{U}} P_e^{us}(x_1^t)$ はモデル $\mathcal{U}$ を用いた時の系列の出現確率の推定値を表している．つまり，式 (11) はパラメータの推定を重み付けにより混合を取っているということである．

以下の式を用いると式 (11) の定義通りの値を再帰的に求めることが出来る．

$$P_w^s(x_1^t) = \begin{cases} \frac{1}{2}P_e^s(x_1^t) + \frac{1}{2}P_w^{0s}(x_1^t)P_w^{1s}(x_1^t) & (s \text{ が文脈木の内部ノード})\\ P_e^s(x_1^t) & (s \text{ が文脈木の葉})\end{cases} \tag{13}$$

ここで, 式 (13) の一行目の式は，右辺第一項 ($\frac{1}{2}P_e^s(x_1^t)$) がノード s 自身，第二項 ($\frac{1}{2}P_w^{0s}(x_1^t)P_w^{1s}(x_1^t)$) がノード s から見た子ノードを表しており，この関係を文脈木の葉の部分まで辿っていくことで重み付け確率を計算することができる．

また $\frac{1}{2}P_e^s(x_1^t)+\frac{1}{2}P_w^{0s}(x_1^t)P_w^{1s}(x_1^t)$ を一般化し $\gamma P_e(x_1^t)+(1-\gamma)P_w^{0s}(x_1^t)P_w^{1s}(x_1^t), (0\leq\gamma\leq 1)$ とする方法 [7] があるが，本研究では 2 値の木情報源を最適に符号化する方法として [1] によって導出された式 (13) を用いる．

式 (13) で計算される P_w^λ は $P(x_1^t|x_{-\infty}^0)$ を推定するものである．つまり，t が大きくなると長い系列の $P(x_1^t|x_{-\infty}^0)$ を推定することになり値が小さくなる．算術符号化で条件付確率を算出する際に，$P(x_1^t|x_{-\infty}^0)$ の推定値と $P(x_1^{t-1}|x_{-\infty}^0)$ の推定値の商を計算する必要があり，計算精度が足りなくなる可能性がある．この問題に対処する方法として Sadakane, Okazaki and Imai が提案した方法 [7] がある．この方法は，新たに文脈 s に対応する β_s という値を導入し，$P(x_t|x_{-\infty}^{t-1})$ を推定するものである．ここで，β_s は以下のように定義される．

$$\beta_s(x_1^t)\stackrel{\text{def}}{=}\frac{P_e^s(x_1^t)}{P_w^{0s}(x_1^t)P_w^{1s}(x_1^t)} \tag{14}$$

この β_s を導入することで以下のように符号化に使用する推定確率を求めることができる．

符号化に使用する推定確率は $P_w^\lambda(x_t|x_1^{t-1})$ であり，一般に $x_{-\infty}^{t-1}$ の接尾辞 s に対して以下の式で定義する．

$$P_w^s(x_t|x_1^{t-1})\stackrel{\text{def}}{=}\frac{P_w^s(x_1^t)}{P_w^s(x_1^{t-1})} \tag{15}$$

s が $x_{-\infty}^{t-1}$ の接尾辞ならば $0s$ または $1s$ も $x_{-\infty}^{t-1}$ の接尾辞である．従って，$k=0,1$ に対し

$$P_w^{ks}(x_1^t)=\begin{cases}P_w^{ks}(x_t|x_1^{t-1})\cdot P_w^{ks}(x_1^{t-1}) & (ks\text{ が }x_{-\infty}^{t-1}\text{の接尾辞})\\ P_w^{ks}(x_1^{t-1}) & (ks\text{ が }x_{-\infty}^{t-1}\text{の接尾辞でない})\end{cases} \tag{16}$$

が成り立つ．よって ks が $x_{-\infty}^{t-1}$ の接尾辞であるとすると式 (13) は以下のように表せる．

$$P_w^s(x_1^t)=\frac{1}{2}P_e^s(x_1^t)+\frac{1}{2}P_w^{ks}(x_t|x_1^{t-1})\cdot P_w^{0s}(x_1^{t-1})\cdot P_w^{1s}(x_1^{t-1}) \tag{17}$$

ここで，β_s と式 (10)(17) を用いると式 (15) は以下のように表すことができる．

$$\begin{aligned}P_w^s(x_t|x_1^{t-1})&=\frac{P_w^s(x_1^t)}{P_w^s(x_1^{t-1})}\\&=\frac{P_e^s(x_1^t)+P_w^{0s}(x_1^t)P_w^{1s}(x_1^t)}{P_e^s(x_1^{t-1})+P_w^{0s}(x_1^{t-1})P_w^{1s}(x_1^{t-1})}\\&=\frac{P_e^s(x_t|x_1^{t-1})\cdot P_e^s(x_1^{t-1})+P_w^{ks}(x_t|x_1^{t-1})\cdot P_w^{0s}(x_1^{t-1})\cdot P_w^{1s}(x_1^{t-1})}{P_e^s(x_1^{t-1})+P_w^{0s}(x_1^{t-1})P_w^{1s}(x_1^{t-1})}\\&=\frac{P_e^s(x_t|x_1^{t-1})\cdot\beta_s(x_1^{t-1})+P_w^{ks}(x_t|x_1^{t-1})}{\beta_s(x_1^{t-1})+1}\end{aligned} \tag{18}$$

また，$\beta_s(x_1^t)$ と $\beta_s(x_1^{t-1})$ の関係は定義より以下のように表せる．

$$\begin{aligned}\beta_s(x_1^t) &= \frac{P_e^s(x_1^t)}{P_w^{0s}(x_1^t)\cdot P_w^{1s}(x_1^t)} \\ &= \frac{P_e^s(x_t|x_1^{t-1})P_e^s(x_1^{t-1})}{P_w^{ks}(x_t|x_1^t)\cdot P_w^{0s}(x_1^{t-1})\cdot P_w^{1s}(x_1^{t-1})} \\ &= \frac{P_e^s(x_t|x_1^{t-1})}{P_w^{ks}(x_t|x_1^{t-1})}\beta_s(x_1^{t-1}) \end{aligned} \tag{19}$$

2.6 符号化手順

文脈木の各内部ノードは図 1 で示したように 0 と 1 に分かれている．文脈 s に対応するノードは $0s$ と $1s$ に対応するノードの親と呼ぶ．そして，各ノード s には $a_s \geq 0$ と $b_s \geq 0$ が対応している．この a_s, b_s はそれぞれ文脈 s の次に $0, 1$ が出現した回数をカウントするカウンタである．親ノード s と子供のノード $0s, 1s$ のカウンタは $a_{0s} + a_{1s} = a_s, b_{0s} + b_{1s} = b_s$ を満たす．この文脈木を用いて $P_w^{\lambda}(x_t|x_{-\infty}^{t-1})$ を式 (13) により計算し，その値を符号化確率として算術符号化する．

以下に有限深さ文脈木重み付け法での符号化の逐次的な手順を示す．まず，葉の深さが全て D の完全な文脈木 $\mathcal{T}_D$ を用意する．最初，用意した文脈木の全てのノードにはカウンタ $(a_s, b_s) = (0, 0)$，内部ノードにはカウンタに加えて重み付け確率 $P_w^s = 1$，$\beta_s = 1$ が保存されている．

1. まず $x_{t-1}, x_{t-2}, \cdots$ の順に文脈に対応した子ノードを選び，文脈木を根から文脈に対応した葉まで辿る．

2. デコーダは x_t を知らないため x_t が 0 であると仮定し，k を文脈 s の子ノード ($k = x_{t-|s|-1}$) とする．この時の重み付け確率 $P_w^s(0|x_1^{t-1})$ を式 (13)(18) を用いて以下のように計算する．

$$P_w^s(0|x_1^{t-1}) = \begin{cases} \dfrac{P_e^s(0|x_1^{t-1})\cdot\beta_s(x_1^{t-1}) + P_w^{ks}(0|x_1^{t-1})}{\beta_s(x_1^{t-1}) + 1} & (s \text{ が文脈木の内部ノード}) \\ P_e^s(0|x_1^{t-1}) & (s \text{ が文脈木の葉}) \end{cases} \tag{20}$$

   これを手順 1 とは逆に s が葉から根 ($s = \lambda$) となるまで親ノードを辿り，文脈木を遡りながら算出する．

3. 符号化確率を $\Pr\{x_t = 0\} = P_w^{\lambda}(0|x_1^{t-1})$ として x_t を符号化する．

4. 実際に $x_t = 0$ だった場合，使用した内部ノードで $\beta_s(x_1^t)$ を式 (19) を用いて以下のように更新し，カウンタ a_s を 1 増やす．

$$\beta_s(x_1^t) = \frac{P_e^s(0|x_1^{t-1})}{P_w^{ks}(0|x_1^{t-1})}\beta_s(x_1^{t-1}) \tag{21}$$

5. $x_t = 1$ だった場合，以下のように式 (20) を計算し直す．

$$P_w^s(1|x_1^{t-1}) = \begin{cases} \dfrac{P_e^s(1|x_1^{t-1})\cdot\beta_s(x_1^{t-1}) + P_w^{ks}(1|x_1^{t-1})}{\beta_s(x_1^{t-1}) + 1} & (s \text{ が文脈木の内部ノード}) \\ P_e^s(1|x_1^{t-1}) & (s \text{ が文脈木の葉}) \end{cases} \tag{22}$$

更に，使用した内部ノードで $\beta_s(x_1^t)$ を以下のように更新し，カウンタ b_s を 1 増やす．

$$\beta_s(x_1^t) = \frac{P_e^s(1|x_1^{t-1})}{P_w^{ks}(1|x_1^{t-1})}\beta_s(x_1^{t-1}) \tag{23}$$

以上の手順 1〜5 を次のシンボルが無くなるまで行う．

3 無限深さ文脈木重み付け法

3.1 無限深さへの拡張と符号化手順

有限深さ文脈木重み付け法は x_{t-D}^{t-1} の文脈のみを見て次のシンボルを推定する．これに対し，Willems が提案した無限深さ文脈木重み付け法 [2] は $x_{-\infty}^{t-1}$ を見て次のシンボルを予測する．このとき，$x_{-\infty}^0$ を知ることが出来ない．そのため Willems は ϵ を不確定なシンボルとし，$x_{-\infty}^0 = \cdots\epsilon\epsilon$ とした．つまり，$x_{-\infty}^0 = \cdots\epsilon\epsilon$ の下で出現したシンボルもカウントし，推定を行っている．これに対し三澤は $x_{-\infty}^0 = \cdots\epsilon\epsilon$ の下で出現したシンボルはカウントせずパラメータ推定には使用しない，という方法をとった [4]．このことで，カウントを行わない分精度は落ちるが，$\cdots\epsilon\epsilon$ という文脈は一度しか出現しないので系列が進むと無視できる．

有限深さ文脈木重み付け法では文脈木の深さは常に D であったが，無限深さ文脈木重み付け法では文脈木が符号化の進行に伴って変化する．この文脈木を $\mathcal{T}$ とする．$\mathcal{T}$ に含まれるモデル全体を $\mathcal{C}_\mathcal{T}$ とし，$\bar{\mathcal{T}}$ を文脈木 $\mathcal{T}$ に含まれる全てのノードの集合とする．モデル $\mathcal{S} \in \mathcal{C}_\mathcal{T}$ を用いて符号 code(s) を

$$\text{code(s)} = \begin{cases} \lambda & (s \in \mathcal{T}) \\ 0 & (s \in \mathcal{S} \text{ かつ } s \notin \mathcal{T}) \\ 1\text{code(0s)code(1s)} & (s \notin \mathcal{S} \text{ かつ } s \notin \mathcal{T}) \end{cases} \tag{24}$$

と定義する．このとき，code(λ) が表す符号語の長さをモデル $\mathcal{S}$ のコストと呼ぶ．すると文脈木 $\mathcal{T}$ におけるモデル $\mathcal{S}$ のコストは

$$\Gamma_\mathcal{T}(\mathcal{S}) = |\mathcal{S}| - 1 + |\{s : s \in \mathcal{S}, |s| \neq \mathcal{T}\}| \tag{25}$$

と表される．ここで文脈木 $\mathcal{T}$ と $s \in \bar{\mathcal{T}}$ に対して

$$\mathcal{T}|s \stackrel{\text{def}}{=} \{r|sr \in \mathcal{T}\} \tag{26}$$

と定める．$\mathcal{T}|s$ はノード s をルートとみなす文脈木 $\mathcal{T}$ の部分木である．文脈木 $\mathcal{T}|s$ に対してモデルコスト $\Gamma_{\mathcal{T}|s}$ は

$$\sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} = 1 \tag{27}$$

を満たす．式 (27) より，$2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})}$ はモデル $\mathcal{U}$ の重みと解釈できる．モデルコストを用い，文脈木 $\mathcal{T}$ とノード $s \in \bar{\mathcal{T}}$ に対して重み付け確率 P_w^s を

$$P_w^s \stackrel{\text{def}}{=} \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{us}, b_{us}) \tag{28}$$

と定義する．この式から式 (13) が導かれる．つまり重み付け確率の計算法は式 (13) の場合と同じでよい．

無限深さ文脈木重み付け法は最初に深さ 0 の文脈木 $\mathcal{T}$ を用意し，必要に応じノードを作成し枝を伸ばすことで文脈木が成長していく．最初 $\mathcal{T}$ の唯一のノード λ にはカウント $(a_\lambda, b_\lambda) = (0, 0)$，予測確率 $P_e^\lambda = 0$，重み付け確率 $P_w^\lambda = 1$，$\beta_\lambda = 1$ を設定しておく．以下に重み付け確率算出の手順を示す．

1. 重み付け確率の条件付確率の算出方法 1〜5 を行う．

2. 文脈 x_1^{t-1} を見る．その文脈に沿い文脈木をたどる．ノードがない場合はノードを作る．その際，初期値は $(a, b) = (0, 0)$，$P_e = 0$，$P_w = 1$ とする．

3. 次のシンボルがある場合は 1 に戻る．ない場合は終了する．

3.2 セグメント

無限深さ文脈木重み付け法において親と子のノードでカウンタの値が等しい場合が多く存在する．カウンタが等しいということは P_e も等しい．よって子の P_w が分かれば，親の P_w も式 (23) より計算できる．この意味でこれらのノードは等価であると言える．したがってそれらのノードを一つのセグメントという単位で管理し，記憶しておくパラメータを最も親（今後先頭と呼ぶ）のノードの物のみにする [2]．しかし，単に一つにしてしまうと親から子の P_w は計算できるが，セグメントの中に何個のノードがあったか（今後セグメントの長さと呼ぶ），どういう形のパスであったかが分からなくなってしまう． そこで，情報源から出力された系

列を記憶しておき，セグメントの先頭が系列のどのシンボルに対応するか，セグメント長をセグメントごとに記憶しておく．すると，セグメントの先頭のノードに対応したシンボルとそれ以前の系列を参照することで，セグメント内の構造を知ることが出来る．この方法を用いることで非常に長いパスをデータ構造上一つのセグメントとして扱うことができる．この方法は，有限深さ文脈木重み付け法にも適用することができる．しかし，有限深さ文脈木重み付け法は文脈木の大きさを予め決めるので，長いパスが出現することはない．このことから，有限深さ文脈木重み付け法に対して大きな効果は期待できない．

前述した通りセグメント内では先頭のノードのみが値を持てばよいので，カウンタ (a, b)，推定確率 P_e，先頭のノードの重み付け確率 P_w，セグメントの先頭ノードが系列のどのシンボルに対応するかを記憶しておく．セグメントを使用メモリの単位とする．

3.3 文脈木の更新

以下に無限深さ文脈木重み付け法を用い，実際に“1101”という系列を符号化する場合の例を示す．前述した通り，無限深さ文脈木重み付け法は $x_{-\infty}^{t-1}$ を見て次のシンボルを予測するが，このとき $x_{-\infty}^{0}$ を知ることが出来ないため，ϵ を不確定なシンボルとし，$x_{-\infty}^{0} = \cdots\epsilon\epsilon$ とすることを前提とする．

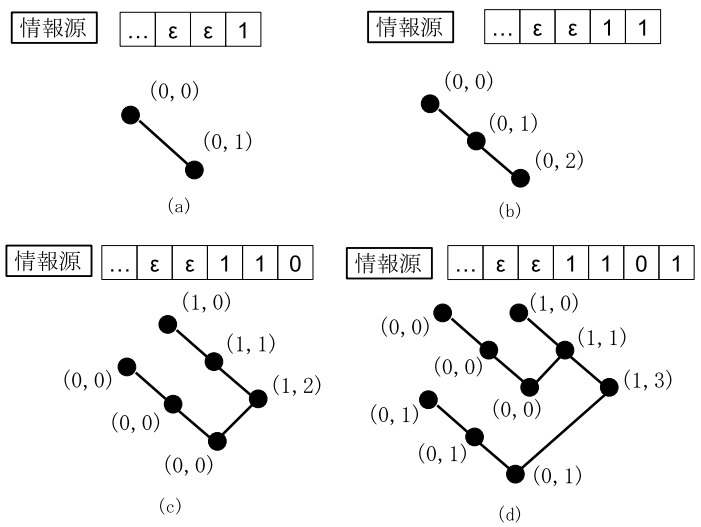

図 3　無限深さ文脈木重み付け法の例

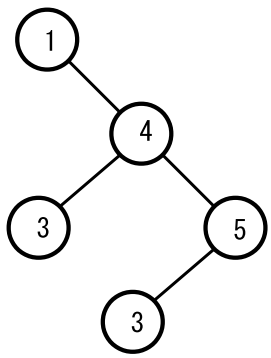

図 4　セグメントの形成

まず，何のシンボルも参照されていない時，文脈木の“根”の部分にあたるノードのみが存在する．このノードのカウンタは前述した通り $(a, b) = (0, 0)$ である．次に情報源から“1”と

いうシンボルが参照されると，カウンタは b が 1 カウントされ，文脈木は図 3(a) のように拡張される．拡張されたノードのカウンタは $(a,b)=(0,0)$ である．次に情報源から“1”というシンボルが参照されると，それぞれのカウンタは b が 1 カウントされ，文脈木は図 3(b) のように拡張される．ここで，参照されたシンボルは「新しく参照されたものから古いものへ見ていく」という点に注意する．同様にして系列が進んでいくと，“1101”というシンボルが参照された時，文脈木は図 3(d) のようになる．

図 3(d) の文脈木をセグメントの形にすると，図 4 のようになる．セグメントの中にある数字は各セグメントの先頭のノードに対応したラベルであり，この先頭のノードを符号化確率の計算に用いる．概念的には図 3(a)(b)(c)(d) のように文脈木を更新するが，これを実現するために，実際にはセグメントを用いて表現された構造を更新する．

3.4 セグメントを用いた逐次的計算

セグメントを用いた場合セグメント内の先頭のノードの値のみ知ることが出来る．また，セグメント内のカウンタ (a,b) が等しいことから P_e は全て等しい．そして P_w は葉から根に向かい再帰的に導出されるものである．つまりセグメントを用いた無限深さ文脈木重み付け法の符号化確率を逐次的に計算するために必要な事は，セグメントの先頭ノードの β から，同セグメント内の全てのセグメントの β を導くことである．以下で導出を行う．

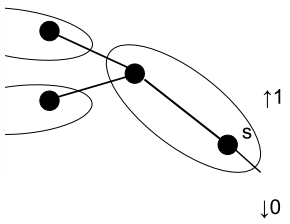

図 5　セグメント内のノード同士の β の関係

図 5 の状況を考える．ノード s はセグメントの先頭ノードを表している．式 (14) より β_s，β_{1s} はそれぞれ

$$\beta_s = \frac{P_e^s}{P_w^{0s}P_w^{1s}} \tag{29}$$

$$\beta_{1s} = \frac{P_e^{1s}}{P_w^{10s}P_w^{11s}} \tag{30}$$

と表せる．

式 (29) と (30) より，β_s と β_{1s} は次のような関係をもつ．

$$\begin{aligned}\beta_s &= \frac{P_e^s}{P_w^{0s} \cdot P_w^{1s}} \\ &= \frac{P_e^{1s}}{\frac{1}{2}P_e^{1s} + \frac{1}{2}P_w^{10s}P_w^{11s}} \\ &= \frac{2\frac{P_e^{1s}}{P_w^{10s}P_w^{11s}}}{\frac{P_e^{1s}}{P_w^{10s}P_w^{11s}} + 1} \\ &= \frac{2\beta_{1s}}{\beta_{1s}+1}\end{aligned} \tag{31}$$

これを β_{1s} について解くと，

$$\beta_{1s} = \frac{\beta_s}{2-\beta_s} \tag{32}$$

となり，セグメントの先頭のノードの子供の β_{1s} が親の β_s の関数で表せた．同様に同一セグメント内であれば β_{0s} を β_s の関数で表すことが出来る．このことから，セグメントの先頭のノードの β から子の β を算出することができ，従来の逐次的計算方法をセグメントを用いた無限深さ文脈木重み付け法に適用できることを証明できた．

3.5 セグメントの消去

大きなファイルを圧縮するには多くのメモリが必要である．そこで，メモリ使用可能量を制限し，上限に達したら更新が一番古いセグメントのメモリを消去するようにする．更新の古いセグメントであるということはあまり使われないセグメントであり，圧縮において瑣末なパスであると言える．このことから，このセグメントは圧縮率への影響が少ないと予想でき，削除することでその文脈が出現しなかったことにしても圧縮率には影響が少ないと考えられる．一番古いセグメントのメモリを消去することは，そのセグメントに対応した文脈が出現したことを忘れることに相当する．つまり，消去するセグメントから根まで全てのカウンタから消去するセグメントのカウンタの値を引くことになる．この方法を用いると，メモリを一つ消去することで，消去したセグメントの兄弟と親のセグメントを統合することができ，結果的に 2 メモリ消去される場合がある事に注意しなくてはいけない．

今，図 6(a) の (I) のセグメントを削除することを考える．(I) のセグメントが削除される際，(Ⅱ) のカウンタは (I) との差になる．すると，(Ⅱ) と (Ⅲ) のセグメントのカウンタが等しくなり，図 6(b) のように統合することができる．結果として 2 メモリ消去されたことになる．この例の場合，(I) のセグメントのカウンタは $(0,0)$ であるため，(Ⅱ) のセグメントのカウンタとの差をとっても (Ⅱ) のカウンタ自体には変化がない．しかし，(I) のセグメントが削除されることにより (Ⅱ) と (Ⅲ) のセグメントは分離している必要がなくなるため，図 (b) のように結合される．

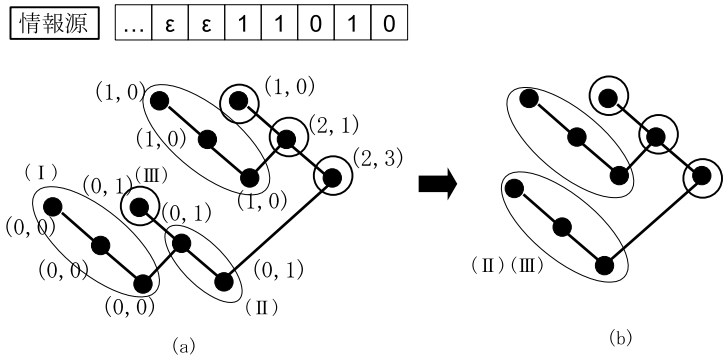

図 6　セグメントの削除と結合

しかし，このようにセグメントの構造が変化するのは消去するセグメントの兄弟と親だけであり，葉から根までの全てのカウンタを操作したとして圧縮率への影響は少ないと考えられる．よって，本研究ではカウンタの操作を行うのは消去するセグメントの親のみとする．

4　算術符号化を用いた無限深さ文脈木重み付け法

三澤は無限深さ文脈木重み付け法において，セグメント数の上限を設定し，上限に達したら更新が最も古いセグメントを消去し上限を超えないようにする方法 [4] を提案した．しかし前述した通り [4] では符号化を行っておらず，符号化確率から符号語長を近似計算している．そこで本研究では実際に算術符号化を行った結果に基づいて圧縮率を算出し，[4] との比較を行う．

圧縮率は [3][4][5] と同様に 1 シンボルあたりの符号語長 [bit/シンボル] と定義する．なお，圧縮を行うファイルとしてカルガリーコーパス [8] を用いる．ファイルの 8[bit] を 1 シンボルとみなす．

4.1　結果

近似計算を行った場合と算術符号化を用いた場合において，生成される最大セグメント数を変化させ，カルガリーコーパスの paper4 と geo というファイルを圧縮した結果を図 7，8 に示した．ここで，最大セグメント数の対数は文脈木の深さに相当するので，横軸の最大セグメント数は対数スケールで表している．また，縦軸は圧縮率を表している．

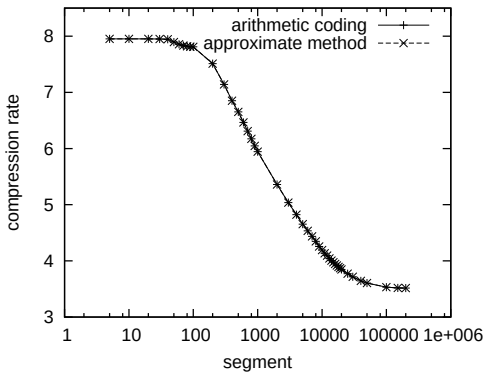

図 7　2 つの方法の圧縮率比較 (paper4)

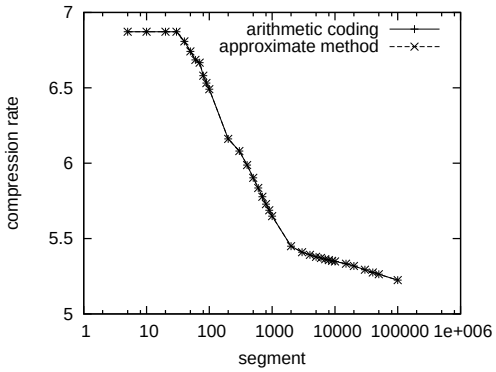

図 8　2 つの方法の圧縮率比較 (geo)

4.2　考察

近似計算を行った方法と算術符号化を行った方法を比較すると，図 7，8 ともに差異はほぼ見られないことがわかる．実際，互いの圧縮率の差の最大値は 0.000137(paper4)，0.000027(geo) であった．すなわち数字の面からも 2 つの方法に差異が見られないことが確認できた．以上より，両方法による圧縮率の計測値の間にはほぼ差がないと結論できる．

5　各セグメントの更新頻度に関する考察

本研究では更新の古いセグメントの消去を行っている．あまり更新されずに消去されるセグメントは系列の進行に対してあまり重要ではないセグメントであり，更新が多く行われるセグメントは系列の中でよく利用されているセグメントである．そこで，この章ではセグメントの更新頻度を調べることによって，消去されるセグメントの特徴を確認する．更新頻度を調べる対象は，符号化の途中で消去されたセグメントと，符号化が終了した時点で残っている全てのセグメントである．更新頻度はセグメントのカウンタ“a”，“b”の値の和を見ればよい．

5.1　結果

生成される最大セグメント数を一定としてカルガリーコーパスのファイルを圧縮し，更新頻度を計測した．その結果を図 9〜14 に示す．横軸 (count) は更新頻度であり，対数スケールを用いている．縦軸 (cumulative frequency) は更新頻度を累積して正規化したものである．また，系列の途中で消去されたものを“delete”，全てのセグメントのものを“all”として示して

ある．paper4，geo，progl，bib のファイルについて最大セグメント数 1000 の場合を測定し，paper4 と geo については最大セグメント数 30000 の場合も測定した．

横軸が対数スケールである理由は次のとおりである．文脈木のセグメント数は深さに対して指数的に増える傾向がある．したがってセグメントの更新頻度は深さに対して指数的に小さくなる．この偏りを補正するために対数スケールを用いている．

5.2 考察

図 9〜14 のグラフは累積頻度を表しているため，傾きが大きいところが更新頻度が大きいことを表している．まず全ての結果に共通して言えることは，消去されるセグメントは更新頻度が小さいということである．普通文脈木においては，根に近いノードほど文脈が出現しやすく，更新頻度は高い値をとる．更新頻度が低い値をとるということは，文脈木において葉に近いセグメントほど消去が頻繁に行われているということが考えられる．

ここで，根に近い方の“必要なセグメント”と，葉に近い方の“不要なセグメント”で線引きが出来ないかを考える．図 11 ではわかりづらいが，図 9，13，14 を見てみると，それぞれ count の数値が 3000(図 9，13)，30000(図 14) のところでグラフの上がり幅が段差のようになっている点がある．ここで，この段差の点より左側ではセグメントの消去が頻繁に行われ，右側ではあまり消去が行われていない，という見方ができる．すなわち，極論ではあるが段差の右側を“必要なセグメント”，左側を“不要なセグメント”として見ることができ，グラフが“二極化”していると考えることができる．先述した通り，段差の左側はカウンタの値が小さい作られたばかりのセグメントが多いため，セグメントの生成回数を減らすことで重み付け確率を算出する際にカウンタの値が大きいものを使用することができる．以上のことから，セグメントの生成回数を減らすことで，プログラムの実行速度の上昇，圧縮率の増加につながるのではないかと考えられる．

しかしながら，先述した通り図 11 では段差がわかりづらいことから，この考え方が適用できる情報源と適用できない情報源が存在することが考えられる．また，図 9，13 と図 14 では段差の点 (横軸 count の値) が異なっており，情報源によって線引きの位置がある一定数に固定できないという欠点につながる．以上のことは十分検討の余地があり，今後の課題として挙げられる．

次に，同じファイルで最大セグメント数のみ変化させた場合について比較する．図 9，10 より，最大セグメント数が多い方が段差の点が左側に寄っていることがわかる．これは，セグメント数が多いためにカウンタの数値がばらけているためであるが，グラフに二極化は存在するため上記の考え方は変わらず適用できると考えられる．しかし，この場合も段差の点が変わっていることから，設定したセグメント数に応じて線引きの位置を適切に変更する手段が必要になる．この検討に関しても今後の課題として挙げられる．

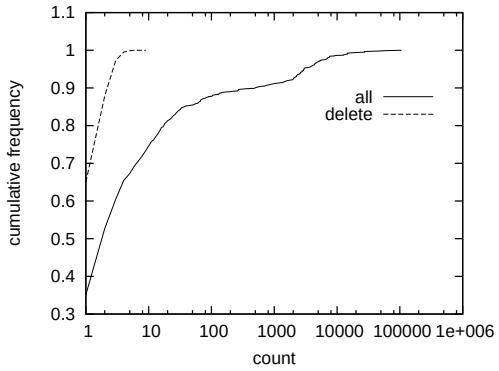

図 9　カウンタの更新頻度 (paper4，セグメント数 1000)

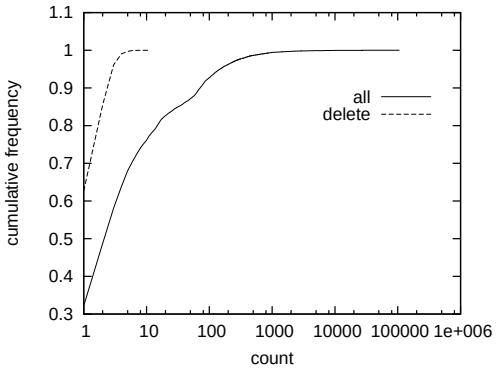

図 10　カウンタの更新頻度 (paper4，セグメント数 30000)

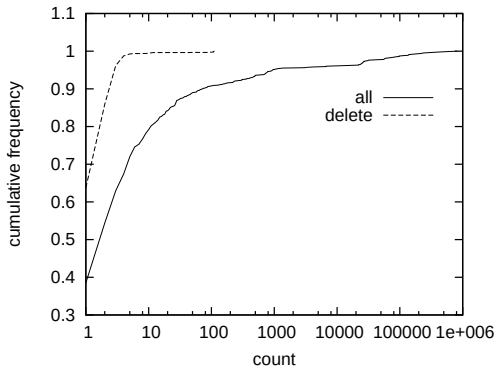

図 11　カウンタの更新頻度 (geo，セグメント数 1000)

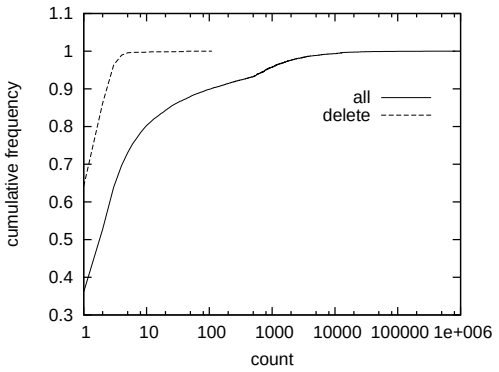

図 12　カウンタの更新頻度 (geo，セグメント数 30000)

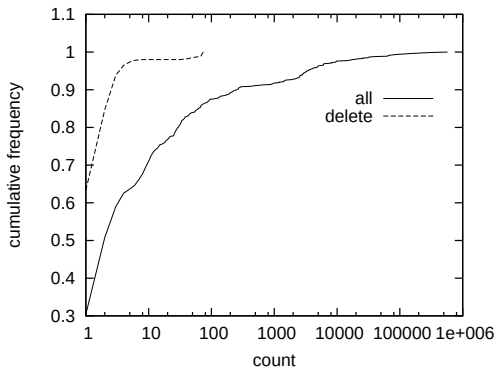

図 13　カウンタの更新頻度 (progl，セグメント数 1000)

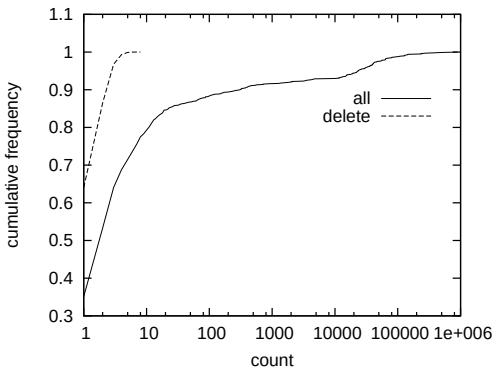

図 14　カウンタの更新頻度 (bib，セグメント数 1000)

6　まとめ

[4] で提案された無限深さ文脈木重み付け法に算術符号化の考え方を取り入れ，比較を行った．その結果，2 つの方法に圧縮率の差異はほとんど見られないことがわかり，近似計算の有効性の程度が確認された．

次に，無限深さ文脈木重み付け法において符号化が終了した時点における全てのセグメントと，消去されたセグメントの更新頻度を調べ，不要なセグメントが存在する可能性について検討した．その結果，セグメントの更新頻度に“二極化”が確認でき，不要なセグメントの存在が示唆された．しかしながら，使用するファイルによっては二極化の点の線引きがしずらいものがあり，この考え方が適用できる情報源と適用できない情報源が存在するものと考えられる．また，使用するファイルや設定する最大セグメント数によって二極化の点の線引きが変わってしまうため，このことについても検討が必要である．

謝辞

本研究を行うにあたって, 細かく指導してくださった指導教員の西新幹彦准教授に感謝の意を表する．

参考文献

[1] Frans M. J. Willems, Yuri M. Shtarkov, Tjalling J. Tjalkens, “The Context-Tree Weighting Method: Basic Properties,” IEEE Transactions on Information Theory, vol.41, no.3, pp.653–664, 1995.

[2] Frans M. J. Willems, “The Context-Tree Weighting Method: Extensions,” IEEE Transactions on Information Theory, vol.44, no.2, pp.792–798, 1998.

[3] 三澤陽一, “有限メモリで動作する文脈木重み付け法の実現,” 信州大学工学部学士論文, 2009.

[4] 三澤陽一, “有限メモリで動作する無限深さ文脈木重み付け法の実現,” 信州大学大学院工学系研究科修士論文, 2011.

[5] 林稔章, “ノードの再帰順位に基づく有限深さ文脈木重み付け法の実現,” 信州大学工学部学士論文, 2012.

[6] David J. C. Mackay, Information Theory, Inference and Learning Algorithms, CAMBRIDGE, 2006.

[7] Kunihiko Sadakane, Takumi Okazaki, Hiroshi Imai, “Implementing the Context-Tree

Weighting Method for Text Compression," Data Compression Conference(DCC'00), pp123–132, 2000.

[8] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus，2013 年 9 月閲覧.

付録 A　算術符号化を用いた圧縮のプログラム

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>


int max_segments;                                       /* セグメント数の上限 */
long number_of_segments;                                /* 現在のセグメント数 */

int codeword_length = 0;

#define Code_value_bits 16
typedef long code_value;

#define Top_value (((long)1 << Code_value_bits) - 1)

#define First_qtr (Top_value / 4 + 1)
#define Half (2 * First_qtr)
#define Third_qtr (3 * First_qtr)
int buffer;
int bits_to_go;
long low;
long high;
long bits_to_follow;

#define SEQUENCE_SIZE 10000000                          /* 出現した系列を保存しておく配列 */
char sequence[SEQUENCE_SIZE];

#define LENGTH 10000000
int size[LENGTH];



typedef struct segment segment;
struct segment{
    long a;                                             /* '0' のカウンタ */
    long b;                                             /* '1' のカウンタ */
    double pe;                                          /* 予測確率 */
    double pw;                                          /* 重み付け確率 */
    double tpe;                                         /* 仮予測確率 */
    double tpw;                                         /* 仮重み付け確率 */
    double log_beta;                                    /* log βの値 */
    segment *parent;                                    /* 親 */
    segment *zero;                                      /* 子供 0 側 */
    segment *one;                                       /* 子供 1 側 */
    int index;                                          /* セグメントの先頭のノードが系列のどこを指しているか */
    long length;                                        /* セグメントの長さ（∞の場合は length = 0） */
    segment *newer;                                     /* 次の要素 */
    segment *older;                                     /* 前の要素 */
};

segment *root;                                          /* ルートへのポインタ */
segment *manage_newest;                                 /* 最新のセグメントへのポインタ */
segment *manage_oldest;                                 /* 最古のセグメントへのポインタ */



segment *alloc_segment()
{
    segment *temporary;

    temporary = (segment *)malloc(sizeof(segment));
    if (temporary == NULL){
        printf("malloc() error in %d\n", __LINE__);
```

```
        exit(1);
    }
    temporary->a = 0;
    temporary->b = 0;
    temporary->pe = 1.0;
    temporary->pw = 1.0;
    temporary->tpe = 1.0;
    temporary->tpw = 0.5;
    temporary->log_beta = 0.0;
    temporary->parent = NULL;
    temporary->zero = NULL;
    temporary->one = NULL;
    temporary->index = 0;
    temporary->length = 0;
    temporary->newer = NULL;
    temporary->older = NULL;

    number_of_segments++;

    return(temporary);
}




double beta_calculate(int times, double log_beta)           /* セグメントの先頭以外のβを計算 */
{
    for (times; times >= 1; times--){
        if (log_beta > 0.0){
            log_beta = -log10((2.0 * pow(10.0, -log_beta)) - 1.0);
        } else {
            log_beta = log_beta - log10(2.0 - pow(10.0, log_beta));
        }
    }

    return(log_beta);
}




double pw_calculate(int temporary_index, segment *current_position, int length)
{
    double temporary_log_beta;
    int times;
    int times_beta;
    double pw;

    times_beta = current_position->length - 1;
    temporary_log_beta = beta_calculate(times_beta, current_position->log_beta);

    if (temporary_log_beta > 0.0){
        pw = 1.0 / (pow(10.0, -temporary_log_beta) + 1.0) * current_position->pe
            + (pow(10.0, -temporary_log_beta) / (1.0 + pow(10.0, -temporary_log_beta))
            * ((sequence[temporary_index - length] == '0')? current_position->zero->pw: current_position->one->pw));
    }else{
        pw = pow(10.0, temporary_log_beta) / (pow(10.0, temporary_log_beta) + 1.0) * current_position->pe
            + (1.0 / (1.0 + pow(10.0, temporary_log_beta))
            * ((sequence[temporary_index - length] == '0')? current_position->zero->pw: current_position->one->pw));
    }

    for (times = current_position->length - length - 1; times > 0; times--){
        times_beta--;
        temporary_log_beta = beta_calculate(times_beta, current_position->log_beta);
        if (temporary_log_beta > 0.0){
            pw = 1.0 / (pow(10.0, -temporary_log_beta) + 1.0) * current_position->pe
                + (pow(10.0, -temporary_log_beta) / (1.0 + pow(10.0, -temporary_log_beta))* pw);
        }else{
```

```
            pw = pow(10.0, temporary_log_beta) / (pow(10.0, temporary_log_beta) + 1.0) * current_position->pe
                + (1.0 / (1.0 + pow(10.0, temporary_log_beta)) * pw);
        }
    }

    return(pw);
}


void listupdate(segment *current_position)                  /* セグメント使用順リスト更新 */
{
    if (manage_newest == current_position){
        return;
    }

    current_position->newer->older = current_position->older;
    if (manage_oldest == current_position){
        manage_oldest = current_position->newer;
    } else {
        current_position->older->newer = current_position->newer;
    }
    current_position->older = manage_newest;
    manage_newest->newer = current_position;
    manage_newest = current_position;
    current_position->newer = NULL;

    return;
}



void dummyupdate(char update)
{
    segment *current_position = root;                       /* 現在見ているセグメント（最初はルート）*/
    int times;                                              /* 計算の回数 */
    double temporary_log_beta;                              /* β一時保管 */
    double temporary_tpw;                                   /* tpw 一時保管 */

    while (current_position->length > 0){
        current_position = ((sequence[current_position->index - current_position->length + 1] == '0')
            ? current_position->zero: current_position->one);
    }
    /* 葉のセグメントの予測確率を計算 */
    current_position->tpe = (((update == '0')? current_position->a: current_position->b) + 0.5)
        / (current_position->a + current_position->b + 1.0);
    if (current_position->tpe < 1e-300){
        current_position->tpe = 1e-300;
    }
    current_position->tpw = current_position->tpe;

    temporary_tpw = current_position->tpw;
    if (update == '0'){
        listupdate(current_position);
    }

    while (current_position->parent != NULL){
        current_position = current_position->parent;

        /* 現在のセグメントの予測確率を計算 */
        current_position->tpe = (((update == '0')? current_position->a: current_position->b) + 0.5)
            / (current_position->a + current_position->b + 1.0);
        times = current_position->length - 1;
        for (times; times >= 0; times--){
            temporary_log_beta = beta_calculate(times, current_position->log_beta);
            if (temporary_log_beta > 0.0){
                temporary_tpw = 1.0 / (pow(10.0, -temporary_log_beta) + 1.0) * current_position->tpe
                    + (pow(10.0, -temporary_log_beta) / (1.0 + pow(10.0, -temporary_log_beta))* temporary_tpw);
            } else {
                temporary_tpw = pow(10.0, temporary_log_beta) / (pow(10.0, temporary_log_beta) + 1.0)
```

```
                    * current_position->tpe + (1.0 / (1.0 + pow(10.0, temporary_log_beta)) * temporary_tpw);
            }
        }

        current_position->tpw = temporary_tpw;

        if (update == '0'){
            listupdate(current_position);
        }
    }

    return;
}




void actualupdate (int current_index, char t)
{
    segment *current_position = root;                       /* 現在位置（最初はルート） */
    segment *child_position;                                /* β計算時に使用 */
    double temporary_log_beta;                              /* β一時保管 */
    int times;                                              /* 計算回数 */


    for (current_index; current_index >= 0 ;current_index --){
        current_position->pe = current_position->tpe;
        current_position->pw = current_position->tpw;

        if (t == '0'){
            current_position->a++;
        } else {
            current_position->b++;
        }

        if (current_position->length != 0){
            times = current_position->length - 1;

            /* セグメント末尾のβを計算 */
            child_position = ((sequence[current_position->index - current_position->length + 1] == '0')
                ? current_position->zero: current_position->one);
            temporary_log_beta = beta_calculate(times, current_position->log_beta);
            temporary_log_beta = temporary_log_beta + log10(current_position->pe / child_position->tpw);
            /* セグメント先頭までβを計算 */
            for (times = current_position->length - 1; times >= 1; times--){
                if (temporary_log_beta > 0.0){
                    temporary_log_beta = log10(2.0) - log10(1.0 + pow(10.0, -temporary_log_beta));
                } else {
                    temporary_log_beta = log10(2.0) + temporary_log_beta - log10(pow(10.0, temporary_log_beta) + 1.0);
                }
            }
            if (temporary_log_beta < -1e+300){
                current_position->log_beta = -1e+300;
            } else {
                current_position->log_beta = temporary_log_beta;
            }
        } else {
            break;
        }
        current_position = ((sequence[current_position->index - current_position->length + 1] == '0')
            ? current_position->zero: current_position->one);
    }

    return;
}
```

```
int trim_confirm(current_index)
{
    segment *current_position;
    int temporary_index;                                    /* index をコピー */
    int number_of_needs;                                    /* セグメントの必要数 */
    int i = 0;

    for (current_position = root; current_index >= 0;){
        temporary_index = current_position->index;
        if (current_position->length == 0 && current_position->index >= 0){
            while (sequence[temporary_index] == sequence[current_index]){
                temporary_index --;
                current_index --;
            }
            if (temporary_index == 0){          /* 現在位置の子供としてセグメントが生まれる場合 (∞ -(1)) */
                number_of_needs = 1;
                break;
            } else {                            /* 現在位置が分裂し、子供もできるばあい (∞ -(2)) */
                number_of_needs = 2;
                break;
            }

        } else {                                                            /* 長さ有限のセグメント */
            while (sequence[temporary_index] == sequence[current_index] && i < current_position->length){
                temporary_index --;
                current_index --;
                i ++;
            }

            if (i == current_position->length){/* 現在位置の子供としてセグメントが生まれる場合 (有限-(1)) */
                if (((sequence[current_index] == '0')? current_position->zero: current_position->one) == NULL){
                    number_of_needs = 1;
                    break;
                } else {
                    current_position = ((sequence[current_index] == '0')
                        ? current_position->zero: current_position->one);
                }

            } else {                                /* 現在位置が分裂し、子供もできるばあい (有限-(2)) */
                number_of_needs = 2;
                break;
            }

        }
    }

    return(number_of_needs);

}


void trim_1(segment *cardinal_position, segment *delete_position)/* 削除したセグメントの親の長さが∞となる */
{
    cardinal_position->a = delete_position->a;
    cardinal_position->b = delete_position->b;
    cardinal_position->pe = 0.5;
    cardinal_position->pw = 0.5;
    cardinal_position->log_beta = 0.0;
    cardinal_position->zero = NULL;
    cardinal_position->one = NULL;
    cardinal_position->length = 0;

    free(delete_position);

    return;
}
```

```
void trim_2(segment *cardinal_position, segment *delete_position, segment *delete_brother)
    /* 兄弟の片方が削除されて親と統合される */
{
    double temporary_log_beta;
    double pw;
    int i;

    cardinal_position->newer->older = cardinal_position->older;
    cardinal_position->older->newer = cardinal_position->newer;
    delete_brother->parent = cardinal_position->parent;
    if (cardinal_position == cardinal_position->parent->zero){
        cardinal_position->parent->zero = delete_brother;
    } else {
        cardinal_position->parent->one = delete_brother;
    }
    delete_brother->index = cardinal_position->index;
    if (delete_brother->length != 0){
        pw = delete_brother->pw;
        temporary_log_beta = delete_brother->log_beta;
        for (i = cardinal_position->length; i > 0; i--){
            if (temporary_log_beta > 0.0){
                temporary_log_beta = log10(2.0) - log10(pow(10.0, -temporary_log_beta) + 1.0);
            } else {
                temporary_log_beta = log10(2.0) + temporary_log_beta - log10(1.0 + pow(10.0, -temporary_log_beta));
            }
            if (temporary_log_beta > 0.0){
                pw = 1.0 / (pow(10.0, -temporary_log_beta) + 1.0) * delete_brother->pe
                    + (pow(10.0, -temporary_log_beta) / (1.0 + pow(10.0, -temporary_log_beta))* pw);
            } else {
                pw = pow(10.0, temporary_log_beta) / (pow(10.0, temporary_log_beta) + 1.0) * delete_brother->pe
                    + (1.0 / (1.0 + pow(10.0, temporary_log_beta)) * pw);
            }
        }
        delete_brother->log_beta = temporary_log_beta;
        delete_brother->pw = pw;
        delete_brother->length = delete_brother->length + cardinal_position->length;
    }
    free(delete_position);
    free(cardinal_position);

    return;
}




void trim_3(segment *cardinal_position, segment *delete_position)
    /* 兄弟の片方が削除されても親と統合されない */
{
    if (delete_position == cardinal_position->zero){
        cardinal_position->zero = NULL;
    } else {
        cardinal_position->one = NULL;
    }
    free(delete_position);

    return;
}




int trim_branches(void)
{
```

```
    int subtrahend_of_segment;                              /* セグメントの減数 */
    segment *delete_position = manage_oldest;               /* 削除するセグメント */
    segment *cardinal_position= delete_position->parent;    /* このセグメントの親 */
    segment *delete_brother = ((cardinal_position->zero == delete_position)
        ? cardinal_position->one: cardinal_position->zero); /* 削除されるセグメントの兄弟 */

    manage_oldest->newer->older = NULL;
    manage_oldest = manage_oldest->newer;

    if (delete_brother == NULL){                            /* 削除するセグメントの兄弟が存在しない場合 */
        trim_1(cardinal_position, delete_position);
        subtrahend_of_segment = 1;

    } else {
        if ((delete_position->a + delete_brother->a == cardinal_position->a)
            && (delete_position->b + delete_brother->a == cardinal_position->b)){
                trim_2(cardinal_position, delete_position, delete_brother);
                subtrahend_of_segment = 2;

        } else {
            trim_3(cardinal_position, delete_position);
            subtrahend_of_segment = 1;

        }
    }

    return(subtrahend_of_segment);
}




void tree_update_1(int current_index, segment *current_position, int times)
{
    segment *temporary;

    temporary = alloc_segment();
    temporary->parent = current_position;
    temporary->length = 0;
    temporary->index = current_index - 1;
    manage_newest->newer = temporary;
    temporary->older = manage_newest;
    manage_newest = temporary;
    if (sequence[current_index] == '0'){
        current_position->zero = temporary;
    } else {
        current_position->one = temporary;
    }
    current_position->length = times;
    return;
}




void tree_update_2(segment *current_position, int temporary_index, int times, int current_index)
{
    segment *temporary_1;
    segment *temporary_2;

    temporary_1 = alloc_segment();
    temporary_1->parent = current_position;
    temporary_1->length = 0;
    temporary_1->index = temporary_index - 1;
    manage_newest->newer = temporary_1;
    temporary_1->older = manage_newest;
    manage_newest = temporary_1;
    if (sequence[temporary_index] == '0'){
```

```
        current_position->zero = temporary_1;
    } else {
        current_position->one = temporary_1;
    }

    temporary_2 = alloc_segment();
    temporary_2->a = current_position->a;
    temporary_2->b = current_position->b;
    temporary_2->pe = current_position->pe;
    temporary_2->pw = current_position->pe;
    temporary_2->log_beta = beta_calculate(times, current_position->log_beta);
    temporary_2->parent = current_position;
    temporary_2->length = 0;
    temporary_2->index = current_index - 1;

    if (current_position == manage_oldest){
        temporary_2->newer = current_position;
        current_position->older = temporary_2;
        manage_oldest = temporary_2;
    } else {
        temporary_2->newer = current_position;
        temporary_2->older = current_position->older;
        current_position->older->newer = temporary_2;
        current_position->older = temporary_2;
    }
    if (sequence[current_index] == '0'){
        current_position->zero = temporary_2;
    } else {
        current_position->one = temporary_2;
    }
    current_position->length = current_position->index - current_index + 1;

    return;
}



void tree_update_3(segment *current_position, int current_index)
{
    segment *temporary;

    temporary = alloc_segment();
    temporary->parent = current_position;
    temporary->length = 0;
    temporary->index = current_index - 1;

    manage_newest->newer = temporary;
    temporary->older = manage_newest;
    manage_newest = temporary;
    if (sequence[current_index] == '0'){
        current_position->zero = temporary;
    } else {
        current_position->one = temporary;
    }

    return;
}




void tree_update_4(int i, segment *current_position, int temporary_index, int current_index)
{
    segment *temporary1;
    segment *temporary2;

    temporary1 = alloc_segment();
```

```
    temporary1->a = current_position->a;
    temporary1->b = current_position->b;
    temporary1->pe = current_position->pe;
    temporary1->log_beta = beta_calculate(i, current_position->log_beta);
    temporary1->parent = current_position;
    temporary1->length = current_position->length - i;
    temporary1->index = temporary_index - 1;
    temporary1->pw = pw_calculate(temporary_index, current_position, temporary1->length);

    temporary1->newer = current_position/*->older->newer*/;
    temporary1->older = current_position->older;
    current_position->older->newer = temporary1;
    current_position->older = temporary1;

    temporary1->zero = current_position->zero;
    temporary1->one = current_position->one;

    if (temporary1->zero != NULL){
        temporary1->zero->parent = temporary1;
    }
    if (temporary1->one != NULL){
        temporary1->one->parent = temporary1;
    }

    if (sequence[current_index] == '0'){
        current_position->one = temporary1;
        current_position->zero = NULL;
    } else {
        current_position->zero = temporary1;
        current_position->one = NULL;
    }
    current_position->length = i;

    temporary2 = alloc_segment();
    temporary2->parent = current_position;
    temporary2->length = 0;
    temporary2->index = current_index - 1;
    manage_newest->newer = temporary2;
    temporary2->older = manage_newest;
    manage_newest = temporary2;
    if (sequence[current_index] == '0'){
        current_position->zero = temporary2;
    } else {
        current_position->one = temporary2;
    }

    return;
}




void tree_update(int current_index, segment *root)
{
    int temporary_index;
    int i;
    segment *current_position;


    for (current_position = root; current_index >= 0 ;current_index --){/* ルートから対応する葉まで辿る */
        int times = 1;                                                   /* βの計算で使用 */
        if (current_position->length == 0 && current_position->index >= 0){
            temporary_index = current_position->index;
            current_position->index = current_index;
            while (sequence[temporary_index] == sequence[current_index]){
                temporary_index --;
                current_index --;
                times ++;
```

```
            }

            if (temporary_index == 0){
                tree_update_1(current_index, current_position, times);
                break;

            } else {
                tree_update_2(current_position, temporary_index, times, current_index);
                break;

            }
        }
        if (current_position->length >= 1){
            temporary_index = current_position->index;
            current_position->index = current_index;
            i = 1;
            while (sequence[temporary_index] == sequence[current_index] && i < current_position->length){
                temporary_index --;
                current_index --;
                i ++;
            }
            if (i == current_position->length){
                if (((sequence[current_index] == '0')? current_position->zero: current_position->one) == NULL){
                    tree_update_3(current_position, current_index);
                    break;

                }else{
                    current_position = ((sequence[current_index] == '0')
                        ? current_position->zero: current_position->one);
                }
            } else {
                tree_update_4(i, current_position, temporary_index, current_index);
                break;

            }
        }
    }
    return;
}



void count_size(segment *child)
{
    if (child != NULL){
        if (child->length == 0){
            size[child->index]++;
        }
        count_size(child->zero);
        count_size(child->one);
    }
    return;
}



void trim_index(segment *child, int subtrahend)
{
    if (child != NULL){
        child->index -= subtrahend;

        trim_index(child->zero, subtrahend);
        trim_index(child->one, subtrahend);
    }
    return;
}
```

```
void trim_sequence(int subtrahend, int index)
{
    int i;
    int j;

    for (i = 1, j = subtrahend + 1; j <= index; i++, j++){
        sequence[i] = sequence[j];
    }
    return;
}




void start_outputing_bits()
{
    buffer = 0;
    bits_to_go = 8;
    return;
}

void output_bit(int bit)
{

    codeword_length++;

    return;
}

void bit_plus_follow(int bit)
{
    output_bit(bit);
    while (bits_to_follow > 0) {
        output_bit(!bit);
        bits_to_follow -= 1;
    }
    return;
}

void start_encoding()
{
    low = 0;
    high = Top_value;
    bits_to_follow = 0;
    return;
}

void encode_symbol(char symbol)     /* 算術符号化 */
{
    long range = (long)(high - low) + 1;
    long bound = low + range * root->tpw;

    if (bound <= low){
        bound = low + 1;
    }
    if (bound >= high + 1){
        bound = high;
    }
    if (symbol == '0'){
        high = bound - 1;
    } else {
        low = bound;
    }

    for (;;){
        if (high < low){
            exit(1);
        }
```

```
        if (high < Half){
            bit_plus_follow(0);
        }
        else if (low >=  Half){
            bit_plus_follow(1);
            low -= Half;
            high -= Half;
        }
        else if (low >= First_qtr && high < Third_qtr){
            bits_to_follow += 1;
            low -= First_qtr;
            high -= First_qtr;
        }
        else break;
        low = 2 * low;
        high = 2 * high + 1;
    }
    return;
}

void done_encoding()
{
    bits_to_follow += 1;
    if (low < First_qtr) bit_plus_follow(0);
    else bit_plus_follow(1);
    return;
}




int main(int argc, char **argv)
{
    char graph_name[100] = {0};
    FILE *graph;

    FILE *srcfp;
    int c;
    int mask;
    int current_index;                                      /* シンボルの出力個数 */
    int number_of_needs;
    int l = 100;
    char t;

    int input_count = 0;


    if (argc != 3) goto ERROR;
    if (argv[2] == NULL) goto ERROR;
    if ((max_segments = (int)atof(argv[2])) == 0.0) goto ERROR;

    start_outputing_bits();
    start_encoding();

    if ((srcfp = fopen(argv[1], "rb")) == NULL){
        printf("fileopen error on %d\n", __LINE__);
        exit(1);
    }

    number_of_segments = 0;

    root = alloc_segment();

    manage_newest = root;
    manage_oldest = root;
```

```
    current_index = 0;
    sequence[current_index] = 'e';
    while ((c = getc(srcfp)) != EOF){
        input_count++;
        for (mask = 1 << 7; mask != 0; mask >>= 1){

            memset(size, 0, sizeof(int) * LENGTH);

            count_size(root);

            dummyupdate('0');                               /* 仮更新 */

            t = (c & mask)? '1': '0';
            encode_symbol(t);

            current_index ++;                               /* シンボルがひとつ出力される */

            if (current_index >= SEQUENCE_SIZE){
                printf("%d\n", __LINE__);
                exit(1);
            }

            sequence[current_index] = t;
            if (sequence[current_index] == '1'){            /* シンボルが 1 だった場合再計算する */
                dummyupdate('1');
            }

            actualupdate(current_index, sequence[current_index]);   /* 本更新 */

            number_of_needs = trim_confirm(current_index);

            while (number_of_segments + number_of_needs > max_segments){
                number_of_segments -= trim_branches();
                number_of_needs = trim_confirm(current_index);
            }

            tree_update(current_index, root);

        }
    }
    done_encoding();

    strcpy(graph_name, argv[1]);
    strcpy(&(graph_name[strlen(graph_name)]), "(提案法).ods");
    if ((graph = fopen(graph_name, "a")) == NULL){
        printf("fileopen error on %d\n", __LINE__);
        exit(1);
    }
    fprintf(graph, "%d,%f\n", (int)atof(argv[2]), (double)codeword_length / input_count);
    fclose(graph);

    fclose(srcfp);

    return(0);

ERROR:
    exit(1);
    return(0);
}
```