信州大学
大学院工学系研究科

修士論文

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

指導教員　西新 幹彦 准教授

専攻　電気電子工学専攻
学籍番号　09TA252H
氏名　三澤 陽一

2011 年 3 月 14 日

目次

1 序論

1.1 研究の背景と目的

現在の私たちの生活に欠かす事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いる．これらの莫大なデータを, そのまま伝送・保存することは非効率的である．この莫大なデータを伝送・保存する際に有効な手段としてデータ圧縮がある．本研究ではその中でも確率モデルに基づいた無歪みデータ圧縮アルゴリズムについて考える．情報源の確率推定と算術符号に基づく方法として 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] がある．文脈木重み付け (Context-Tree Weighting,CTW) 法は, モデルとパラメータが未知である条件のもとで, 木情報源を計算効率良く, 最適に符号化する確率推定法である．確率モデルに基づくデータ圧縮では, 情報源がどういう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．符号化に先立ち, 情報源系列全体を見ることができるならば, それにより確率構造を予め調べることができる．本研究で取り上げる文脈木重み付け法では, 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, 全ての情報がそろ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, 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．そこで本研究では, 生成できるセグメント数に制限を付けることで, ファイルの大きさと必要なメモリの関係をなくし, 有限メモリで動作させる方法について考える．

1.2 本論文の構成

本論文では, 次のような構成をとる．2 章では木情報源,CTW 法について説明する．3 章では有限深さ文脈木重み付け法について説明する．4 章では無限深さ文脈木重み付け法について, 提案法の説明, 結果と評価を示す．5 章ではまとめをする．

2 文脈木重み付け法

有限深さ文脈木重み付け法とは木情報源をパラメータ未知, モデル未知の条件下において, シンボル当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に符号化する確率推定法である．有限深さ文脈木重み付け法の最適性については Willems, Shtarkov and Tjalkens によって証明されている [1]．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方法で, 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．

2.1 木情報源

x_m^n は系列 x_m，x_{m+1}，$\cdots$，x_n を意味している．$n < m$ ならば $x_m^n = \lambda$ と定義する．λ は空系列を意味している．文字列の語尾を接尾辞と言う．x_m^n の接尾辞は λ，x_n^n，x_{n-1}^n，$\cdots$，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であると言う．$|s|$ を文字列の長さを表す演算子とする．また本研究では 2 値の情報源を考える．接尾辞フリーな集合 $\mathcal{S}$ が完全とは $\sum_{s\in\mathcal{S}} 2^{-|s|} = 1$ が成り立つことを言う．

木情報源とは系列 $x_{-\infty}^{\infty}$ を生み出す Markov 情報源の一種であり, 以下のように定義される．完全な接尾辞フリーな有限集合 $\mathcal{S}$ を考える．$s \in \mathcal{S}$ の長さ $|s|$ は $|s| \leq D$ であると仮定する．この D をモデルの深さと呼ぶ．$\mathcal{S}$ に対して完全木が対応し, 各 $s \in \mathcal{S}$ はその葉に対応する．例として $\mathcal{S} = \{00, 10, 1\}$ に対して図 1 のような完全木が対応する．$\mathcal{S}$ に属する各々の文字列 s にパラメータ θ_s が対応する．各パラメータ θ_s は [0,1] の値をとり, {0,1} 上の分布を表す．このようにパラメータ全体はパラメータベクトル $\Theta_{\mathcal{S}} \stackrel{\text{def}}{=} \{\theta_s : s \in \mathcal{S}\}$ を成す．またこの $\mathcal{S}$ を情報源のモデルという．現在までに x_m^n が出現しているとき,D シンボルさかのぼって系列 x_{n-D+1}，x_{n-D+2}，$\cdots$，x_n を見れば, この系列の接尾辞であるような $\mathcal{S}$ の要素 s は一意に決まる．このとき s に対応する θ_s が決まり, θ_s にしたがい x_{n+1} の値が決められる．また, 情報源 X から発生する系列を $x_1x_2\cdots x_t\cdots$ としたとき, 時刻 t における文脈とは x_{t-d}^{t-1} $(0 \leq d \leq D)$ のことである．

2.2 推定器

CTW 法ではパラメータを推定するための推定器は任意のものでよい．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として Krichevsky-Trofimov(KT) 推定が存在する．本研究では [1] に従いこの KT 推定を用いる．‘0’,‘1’ が出現した回数を a,b とすると，KT 推定確率 $P_e(a, b)$ は以下のように定義される．

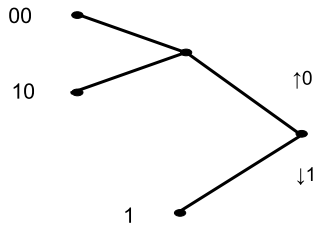

図 1　モデルと完全木

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

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できる.

2.3　モデルのコスト

モデル $\mathcal{S}$ を用いて符号 $\mathrm{code}(s)$ を以下のように定義したとき,$\mathrm{code}(\lambda)$ が表す符号語の長さを $\mathcal{S}$ のコストと呼ぶ.

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

すると $\mathcal{S}$ のコストは次の式で表される.

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

2.4　重み付け確率

全ての葉の深さが D の完全木を文脈木と呼ぶ. 各ノードは文脈に対応する. このとき x_{t-D}^{t-1} は葉に対応する. 文脈 $s = x_{t-d}^{t-1}$ に対応するノードの深さを $d(d \leq D)$ とする. 深さ $D-d$ 以

下のモデル全体を $\mathcal{C}_{D-d}$ とする．また,a_s,b_s を文脈 s の次に ‘0’,‘1’ が出現した回数とする．モデルのコストを用い文脈 s の重み付け確率 P_w^s を次の式で定義する．

$$P_w^s \stackrel{\text{def}}{=} \sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} \prod_{u\in\mathcal{U}} P_e(a_{us}, b_{us}) \tag{7}$$

$$\text{このとき} \sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} = 1 \tag{8}$$

式 (8) はいわゆる等号が成立しているクラフトの不等式 [4] である．よって $\sum_{\mathcal{U}\in\mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})}$ は深さが $D-d$ 以下の全ての完全木への重み付けを意味している．$\prod_{u\in\mathcal{U}} P_e(a_{us}, b_{us})$ はモデル $\mathcal{U}$ を用いたときの系列の出現確率の推定値を表している．つまり, 式 (7) 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ことになる．

以下の式を用いると式 (7) の定義の通りの値を再帰的に求めることができる．

$$P_w^s = \begin{cases} \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}P_w^{1s} & (s \text{ が文脈木の内部ノード}) \\ P_e(a_s, b_s) & (s \text{ が文脈木の葉}) \end{cases} \tag{9}$$

また,$P_w^s = \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}P_w^{1s}$ を一般化し,$P_w^s = \gamma P_e(a_s, b_s) + (1-\gamma)P_w^{0s}P_w^{1s}$,$(0 \le \gamma \le 1)$ とする方法 [5] 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 (9) を用いる．

2.5 文脈木重み付け法の逐次的計算アルゴリズム

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．文脈木の各内部ノードは ‘0’ と ‘1’ に分かれている．文字列 s に対応するノードを ‘$0s$’ と ‘$1s$’ に対応するノードの親と呼ぶ. そして, 各ノード $s \in \mathcal{T}_D$ には $a_s \ge 0$ と $b_s \ge 0$ が対応する．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．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 P_w^λ を式 (9) で計算し, 符号化確率とする．

式 (9) 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 $P(x_1^t|x_{-\infty}^0)$ の推定と $P(x_1^{t-1}|x_{-\infty}^0)$ の推定の商を計算する必要があり, 計算精度が足りなくなる恐れがある．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 [5] がある．この方法は新たに $\beta_s = \frac{P_e^s}{P_w^{0s}P_w^{1s}}$ という値を導入することで，$P(x_t|x_{-\infty}^{t-1})$ を推定するものである．また，β_s を導入することで (9) を条件付確率として後述する (10) に書きかえられる．

以下に x_t の符号化の逐次的の手順を示す．文脈木の各ノードには $x_{-\infty}^{t-1}$ に基づくカウント (a_s,b_s), 予測確率 P_e^s, 重み付け確率 P_w^s, β_s 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を選び, 文脈木を根から文脈に対応した葉までたどる.

2. カウント (a_s, b_s) から予測確率 P_e を算出する. 更に $c = 0$,$k = x_{t-|s|-1}$ として

$$P_w^s = \begin{cases} \frac{\beta_s}{\beta_s+1} P_e^s(x_t = c) + \frac{1}{\beta_s+1} P_w^{ks}(x_t = c) & (s \text{ が文脈木の内部ノード}) \\ P_e^s & (s \text{ が文脈木の葉}) \end{cases} \tag{10}$$

を計算する. これを現在位置から根まで遡りながら算出していく. なおデコーダは x_t を知らないため，x_t が '0' であると仮定して計算する.

3. 符号化確率を $\Pr\{x_t = 0\} = P_w^\lambda$ として実際の x_t を符号化する.

4. $x_t = 1$ だった場合は $c = 1$ として (10) を計算しなおす.

5. 更新したノードで,c は実際の x_t として

$$\beta_s(x_1^t) = \beta_s(x_1^{t-1}) \frac{P_e^s(x_t = c)}{P_w^{ks}(x_t = c|x_1^{t-1})} \tag{11}$$

とし，$x_t = 0$ ならカウンタ a_s を 1 増やし，$x_t = 1$ ならカウンタ b_s を 1 増やす.

3 有限深さ文脈木重み付け法

3.1 符号化手順

葉の深さが全て D の完全な文脈木 $\mathcal{T}_D$ を用い，CTW 法で系列確率を推定するのが有限深さ文脈木重み付け法である．最初文脈木の各ノードにはカウント $(a_s,b_s) = (0, 0)$, 予測確率 $P_e^s = 1$, 重み付け確率 $P_w^s = 1$, $\beta_s = 1$ が保存されている．アルゴリズムは重み付け確率の条件付確率の算出方法 1〜5 を次のシンボルがなくなるまで行う.

3.2 メモリの使用効率を向上させる方法

有限深さ文脈木重み付け法は文脈木の深さ D を決めることで符号化に必要なメモリ数が決まり，D を大きくするとより深い文脈を考慮出来るので圧縮率が良くなる傾向にある．しかし，D を大きくすると出現しない文脈が多く無駄なメモリが出てきてしまう．これに対し, 有限のメモリを効率よく使用する方法 [3] が提案されている．この方法は予めメモリを用意せず

に, ノードを使用したときにメモリを確保するようにし, さらに生成できるノード数に上限を設定する. 上限に達したら更新が一番古いノードのメモリを消去するようにする. これは更新が古いノードに対応する文脈はあまり出現しない文脈であり, 圧縮率への影響が少ないことから提案された. なお削除される際にノードで保存されているカウンタ (a, b) もリセットされる. この削除されるノードの親のカウンタを操作する必要はない. 何故ならノードのカウンタは対応する文脈の次に出現した 0, 1 をカウントするものであり, 子のカウンタを削除し対応する文脈が出現したことを忘れたとしても, 親に対応する文脈が出現したことを忘れる必要がないからである. 削除したことで, そのノードが元々なかったことになるだけなので計算方法に問題は生じない. また提案法は有限深さ文脈木重み付け法を実装する方法であり, 文脈木の深さ D を設定する必要がある. つまり多くのメモリを使用できる状況で, D を小さく設定してしまうと, メモリの削除があまり起きなくなり瑣末なノードのメモリが生き残り, D より深く重要な文脈があったとしてもノードを作ることができない.

図 2 に $x_{1-D}^{0} = \cdots 00$, $x_1^T = 11$, 深さ 2, メモリ上限 5 の例を示す. 実線はノードが存在し枝が伸びていることを示し, 破線はメモリが存在せず枝が伸びていないことを示している. 図 2 の丸数字はノードが使用される順番を表わしている. 1〜6 はメモリ使用量が上限に達していないためメモリを確保することができる. しかし, 7 を確保するときに上限を超えてしまう. そこでもっとも昔に使用したノードに対応したメモリ, つまり 1 のメモリを削除している.

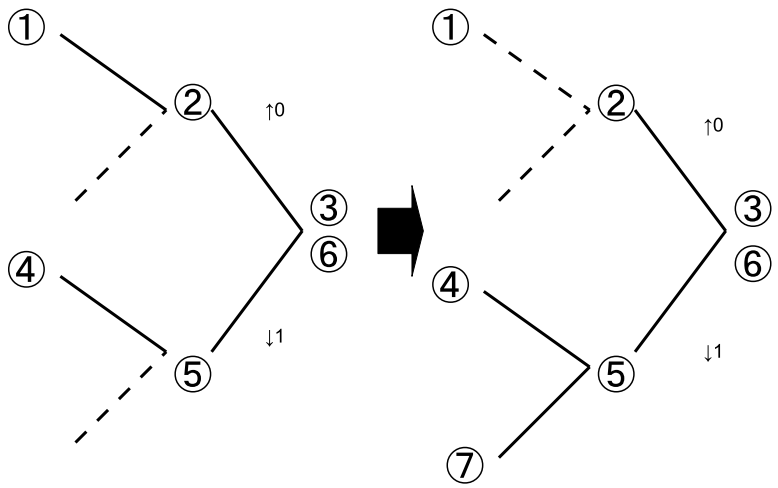

図 2　メモリの削除方法 (メモリ上限 5)

このように最も昔に使用したノードに対応したメモリを削除することで上限を超えないようにできる．この方法を用いることで, 従来の方法で無駄になっていたメモリの分だけ，D を大きくする事で重要な深いノードのメモリを確保することができるようになる．また, 同じメモリ量で符号化した場合, 提案法の文脈木の深さを従来法よりある程度深くすると圧縮率が改善されている [3].

有限深さ文脈木重み付け法と無限深さ文脈木重み付け法において, 更新が古いノードが圧縮率に瑣末であるという点は共通である．よって最も古いノードを削除すると言うメモリの制限方法は, 無限深さ文脈木重み付け法にも有効であると考えられる．

4 無限深さ文脈木重み付け法

4.1 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ので 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．これに対し本研究では $x_{-\infty}^{0} = \cdots\epsilon\epsilon$ 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．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が D であったが, 無限深さ文脈木重み付け法では, 文脈木が系列により変化する．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，$\overline{\mathcal{T}}$ 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 $\mathrm{code}(s)$ を

$$\mathrm{code}(s) = \begin{cases} \lambda & (s \in \mathcal{T}) \\ 0 & (s \in \mathcal{S} \text{ かつ } s \notin \mathcal{T}) \\ 1\mathrm{code}(0s)\mathrm{code}(1s) & (s \notin \mathcal{S} \text{ かつ } s \notin \mathcal{T}) \end{cases} \tag{12}$$

と定義する. このとき,$\mathrm{code}(\lambda)$ 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

$$\Gamma_{\mathcal{T}}(\mathcal{S}) = |\mathcal{S}| - 1 + |\{s : s \in \mathcal{S}, s \notin \mathcal{T}\}| \tag{13}$$

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

$$\mathcal{T}|s \stackrel{\mathrm{def}}{=} \{r|sr \in \mathcal{T}\} \tag{14}$$

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

$$\sum_{\mathcal{U}\in\mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} = 1 \tag{15}$$

を満たす. 証明は付録 A に記載する. (15) 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 $s \in \overline{\mathcal{T}}$ に対して重み付け確率 P_w^s を

$$P_w^s \stackrel{\text{def}}{=} \sum_{\mathcal{U}\in\mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \prod_{u\in\mathcal{U}} P_e(a_{us}, b_{us}) \tag{16}$$

と定義する. この定義から式 (9) が導かれる. つまり重み付け確率の計算法は (9) と同じでよい. この証明は付録 B に記載する.

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 $\mathcal{T}$ 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 へ戻る. ない場合は終了する.

4.2 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物が多く存在する. 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 (11) 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. 等価であるので図 3 のようにそれらのノードを一つのセグメントとして考え, 記憶しておく値を最も親（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．よってセグメント数は文脈木の大きさを直接的には制限しない．またこの方法は有限深さ文脈木重み付け法にも適用できる．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, カウント (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き, これをメモリの単位とする．

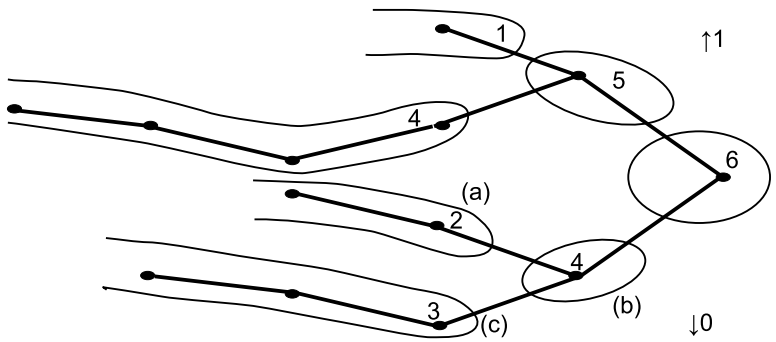

図 3　等価なノードを一つのセグメントとした文脈木 $(x_1^t = 11001)$

4.3　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．以下で先頭の β から, 同セグメント内のそれ以降の β を導出する．

図 4 の状況を考える．ノード s はセグメントの先頭のノードを指している．β_s，β_{1s} はそれぞれ.

$$\beta_s = \frac{P_e^s}{P_w^{0s} P_w^{1s}} \tag{17}$$

$$\beta_{1s} = \frac{P_e^{1s}}{P_w^{10s} P_w^{11s}} \tag{18}$$

と表される．

(17) と (18) より，β_s と β_{1s} の関係を以下のように導く．

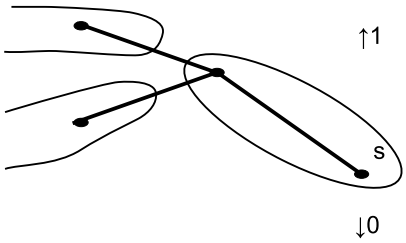

図 4　セグメント内のノード同士の β の関係

$$\beta_s = \frac{P_e^s}{P_w^{0s} P_w^{1s}} \tag{19}$$

$$= \frac{P_e^{1s}}{\frac{1}{2} P_e^{1s} + \frac{1}{2} P_w^{10s} P_w^{11s}} \tag{20}$$

$$= \frac{2P_e^{1s}}{P_e^{1s} + P_w^{10s} P_w^{11s}} \tag{21}$$

$$= \frac{2\frac{P_e^{1s}}{P_w^{10s} P_w^{11s}}}{\frac{P_e^{1s}}{P_w^{10s} P_w^{11s}} + 1} \tag{22}$$

$$= \frac{2\beta_{1s}}{\beta_{1s} + 1} \tag{23}$$

式 (19) の P_e^s は同セグメント内のカウンタが一致していることから $P_e^s = P_e^{1s}$, ノード $0s$ はカウント (0,0) なので $P_w^{0s} = 1$, P_w^{1s} は定義式 (9) を代入する. 式 (20) の分母, 分子を 2 倍する. 式 (21) の分母, 分子を $\frac{1}{P_w^{10s} P_w^{11s}}$ 倍する. 式 (18) を用い, (22) を置き換える. これを β_{1s} について解くと,

$$\beta_{1s} = \frac{\beta_s}{2 - \beta_s} \tag{24}$$

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. 同様に同セグメント内であれば β_{0s} を β_s の関数で表せる. 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 を算出することが出来, 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.

4.4　有限個のセグメントで動作する無限深さ文脈木重み付け法

セグメントを導入した場合でも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．圧縮において瑣末なパスであることから, 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のカウンタを, 消去するセグメントのカウンタとの差にすることになる．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 3 の (a) のセグメントを削除するとする．(a) のセグメントが削除されたことで，(b) のカウンタは (a) との差になる．すると，(b) と (c) のセグメントのカウンタが等しくなり, 図 5 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．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．また消去するセグメントの親のカウンタを操作する場合と, 消去するセグメントから根まで全てのカウンタを操作する場合とで比較を行う事は今後の課題として挙げられる．

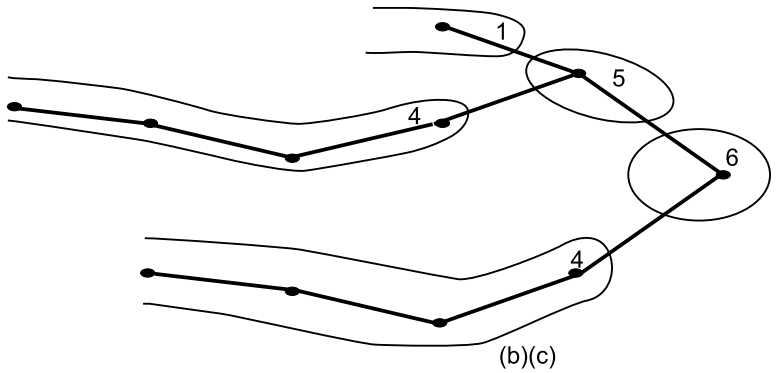

図 5　セグメントの削除と統合

4.5　検証実験

CTW 法で系列確率を推定し, それを符号化確率として算術符号を行う．しかし, 本研究の実験では算術符号を行わずに符号化確率から符号語長を計算する．P を符号化確率，l を符号語長とすると算術符号の符号語長は次のように評価できることが知られている．

$$-\log_2 P + 1 \leq l \leq -\log_2 P + 2 \tag{25}$$

これを情報源のシンボル数 n で割り平均符号語長にすると,

$$\frac{-\log_2 P}{n} + \frac{1}{n} \leq \frac{l}{n} \leq \frac{-\log_2 P}{n} + \frac{2}{n} \tag{26}$$

となる. ここで n は非常に大きい値なので $\frac{1}{n}$ と $\frac{2}{n}$ は無視できる. よって, 本研究では $-\log_2 P$ の値を符号語長として扱う. このとき符号化確率を条件付確率で算出すると 1 シンボル当たりの符号語長を $L = \frac{1}{n}\sum_{i=1}^{n} l_i$ とできる. 本論文における圧縮率を 1 シンボルあたりの符号語長 [bit/シンボル] と定義する. 圧縮率はこの値が小さいほどよいと言える. また, 本研究では圧縮率の検証をする際にカルガリーコーパス [6] を用いる. カルガリーコーパスのファイルを 8[bit] で 1 シンボルとみなす.

提案法を用いカルガリーコーパスを圧縮した結果を図 6〜図 9 に示す. paper4 をセグメント数を制限せずに圧縮すると最終的に 212541 個のセグメントが生成される (図 6 破線). 生成可能セグメント数を 212541 こから約 1/2 の 100000 個にする. しかし, 生成できるセグメント数を 1/2 にした事で圧縮率はほとんど悪くならない. 他の実験結果図 7〜図 9 でも同様のことが言える. つまり最も更新の古いセグメントを消去することでメモリを制限する方法は, 無限深さ文脈木重み付け法に有効であると言える.

4.6 保存系列の削除

提案法を用いると, 前節の例のようにセグメントを統合できる場合が多く出てくる. また, 系列が進むと, 文脈木のパスが長くなる. これらのことから, 提案法を用いると葉のセグメントが長くなる傾向にあると考えられる. 提案法を用い, 最終的に完成した文脈木の葉のセグメント長とその個数の関係を図 10 に示す.

図 10 から, 全ての葉のセグメント長が 40000 より大きいことがわかる. このことは, セグメントの内部構造を知るために保存している系列 x_1^{40000} は最終的に使用されないということである.

そこで任意の閾値を設定する. そして, 最も短い葉のセグメント長が閾値より大きくなったら, 最も短い葉のセグメント長と閾値の差の長さだけより古い系列を削除する. 葉のセグメント長はセグメントが指す系列のシンボルから x_1 の長さになので, 古い系列を削除すると葉のセグメントは系列を削除した分だけ短くなる.

図 11〜14 に閾値と圧縮率の関係を示す. 図 11 から閾値を設定した場合と閾値を設定しなかった場合の圧縮率がほぼ一致していること分かる. なお図 11 では確認できないがこの実験では閾値を設定した場合の方が約 1/10000[bit/シンボル] 程度圧縮率が悪くなっている. つまり, 設定したことで圧縮率がほとんど悪くならないと言える. 図 12〜14 も同様の結果が得られた.

次に系列の削除がどのように起こっているかを確認するために, 提案法を用い閾値を 10 に

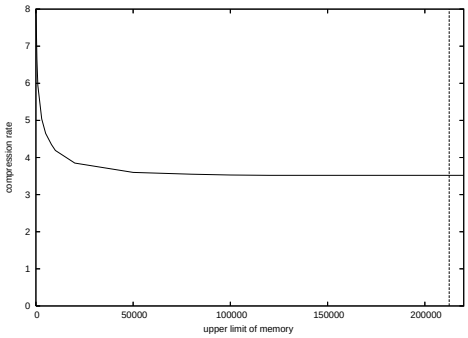

図 6　生成可能セグメント数に対する圧縮率 (paper4)

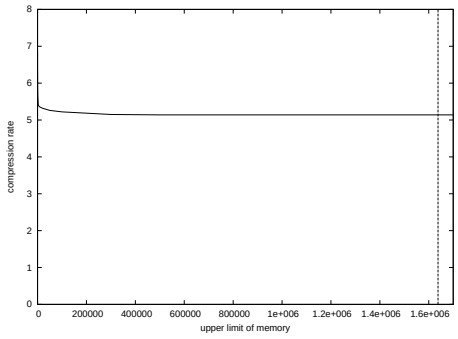

図 7　生成可能セグメント数に対する圧縮率 (geo)

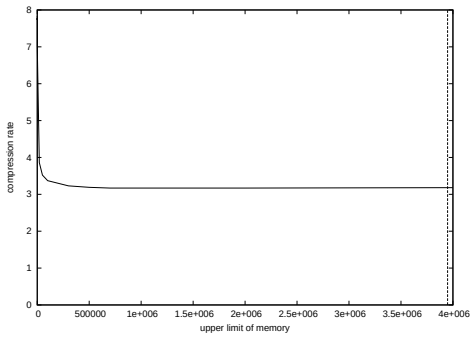

図 8　生成可能セグメント数に対する圧縮率 (obj2)

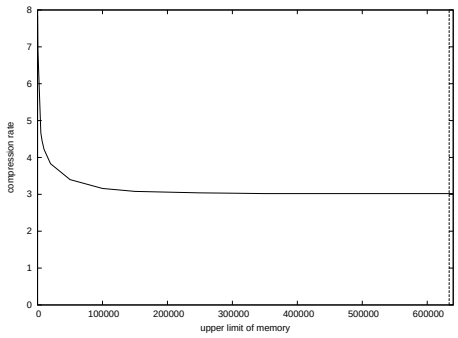

図 9　生成可能セグメント数に対する圧縮率 (progc)

設定し符号化を行った場合の保存系列長の推移を図 15〜18 に示す. 図 15 を見ると, 保存シンボル数は削除が起こるまでは入力数に比例して増えていくが, そこからは振動しているが, 一定量を超えることはない. 図 16〜18 も同様のことが言える. このことから, 実質的に有限で動作していると言える. 図 15〜18 を比較すると, セグメント数上限が大きいほど振動の中心が高いことが分かる. このことから, どこを中心に振動するかは生成できるセグメント数の上限によって異なると考えられる. セグメント数の上限を大きくすると, あまり更新されないセグメントも長く生き残ることになる. このことであまり使用されずにいる短い葉のセグメントも長く生き残る. 短い葉のセグメントが長く生き残る事は, 全ての葉のセグメントが閾値を超えない原因となる. そして系列の削除が遅くなり, 振動する中心が高くなると考えられる. つまり, 保存系列を短くしたければ, 生成できるセグメント数を少なくすれば良いということである.

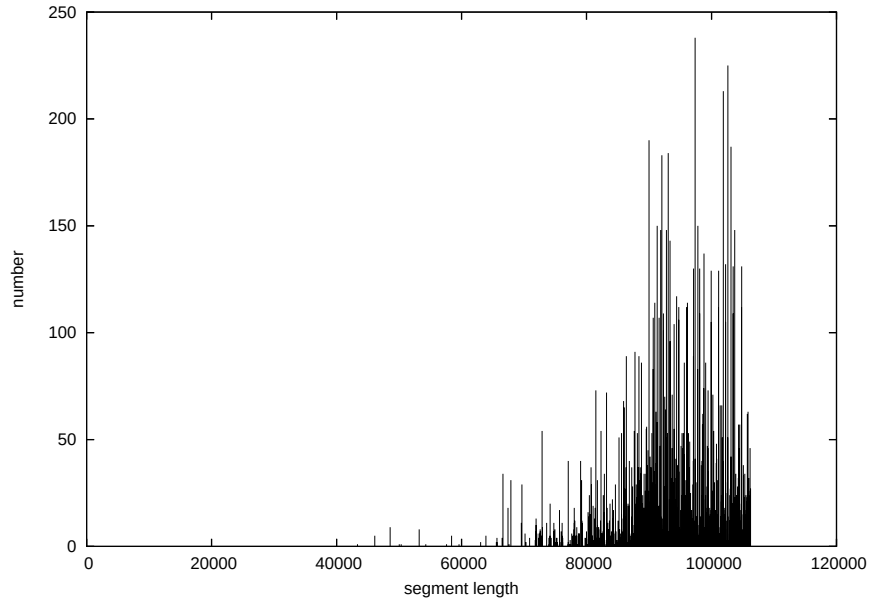

図 10　葉のセグメント長と個数 (paper4)

しかし, 生成できるセグメント数を少なくすると圧縮率が悪くなる (図 6〜9).

発生した系列を古いものから忘れていく方法として文脈木重み付け法を有限窓で動作させる方法が提案されている [7]. 本論文の提案法は古い保存系列長を削除していくという点で有限窓法に似ているが, 同じではない. 有限窓法は固定長の系列を見て符号化を行う. これは圧縮率に重要な情報を忘れてしまう可能性がある. それに対し本論文の提案法は, 必要なくなったであろう系列を削除する. つまり, 圧縮率に瑣末と思われる系列のみを忘れることができるのである.

5　まとめ

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を提案し圧縮率を検証した. 結果として, 圧縮率をあまり悪くせずにセグメント数を制限できた. 次にセグメント数に上限を付け符号化を行った際に, 全ての葉のセグメント長が非常に大きい事を発見した. このことから, 閾値を設定し全ての葉のセグメント長が閾値を超えたら, 保存している系列を削除する方法を提案した. この方法により圧縮率はほとんど悪くならず, また保存系列の削除は頻繁に起こり, 実質的に有限長の系列を保存することで動作させることができた. 今後の課題として, セグメ

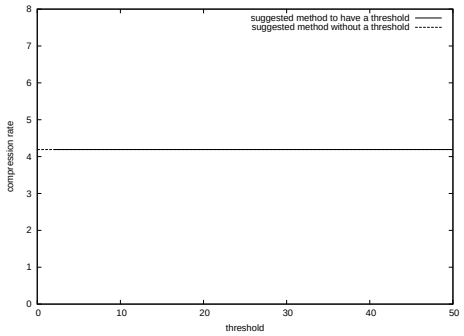

図 11　閾値と圧縮率の関係 (paper4, セグメント数上限 10000)

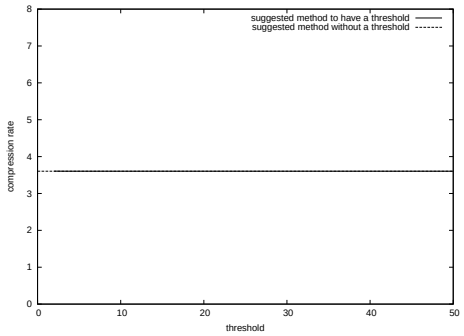

図 12　閾値と圧縮率の関係 (paper4, セグメント数上限 50000)

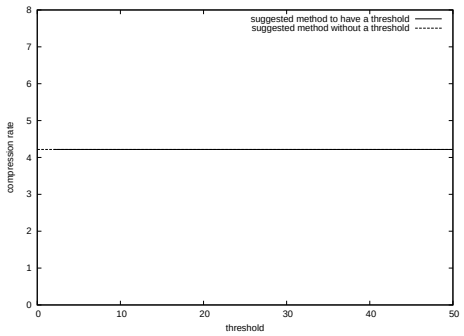

図 13　閾値と圧縮率の関係 (progc, セグメント数上限 10000)

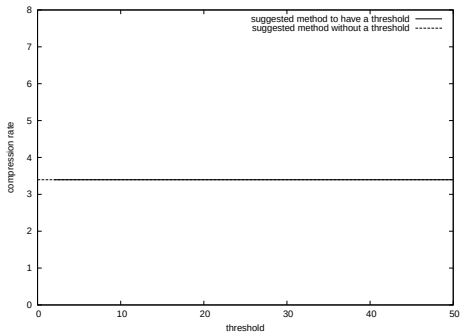

図 14　閾値と圧縮率の関係 (progc, セグメント数上限 50000)

ント数に上限をつけ閾値を設定した場合に, セグメント数上限, 閾値, 情報源の性質が保存系列長の推移にどのように関係しているのかを詳しく調べる必要がある．また, 最も古いセグメントを削除するという方法, 閾値を提案したが, この方法が最も適した方法なのか, 他の方法はないのかということも調べる必要がある．

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．

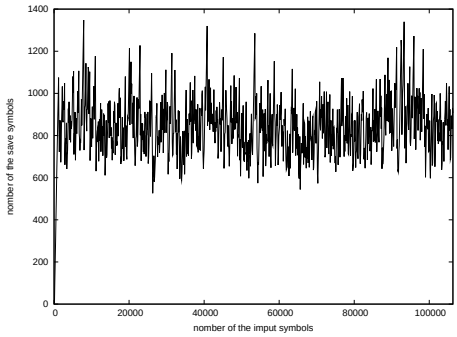

図 15　入力シンボル数に対する保存系列長の推移 (セグメント数上限 1000)

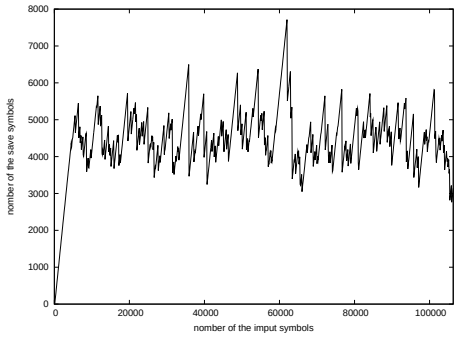

図 16　入力シンボル数に対する保存系列長の推移 (セグメント数上限 5000)

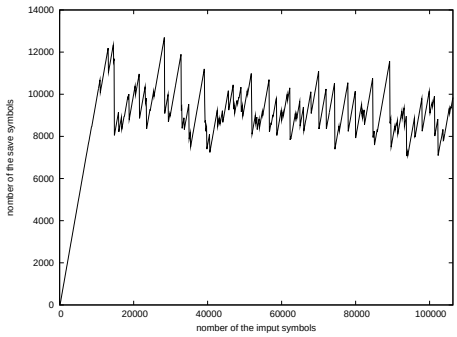

図 17　入力シンボル数に対する保存系列長の推移 (セグメント数上限 10000)

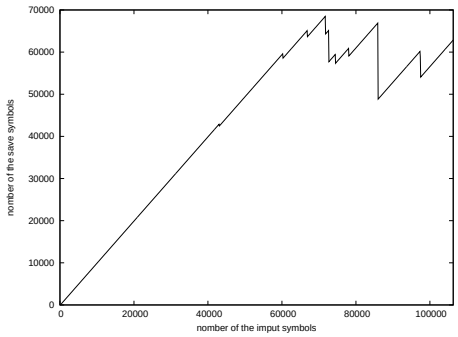

図 18　入力シンボル数に対する保存系列長の推移 (セグメント数上限 50000)

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] David J. C. MacKay, “ Information Theory. Inference, and Learning Algorithms, ” CAMBRIDGE, 2006.

[5] Kunihiko Sadakane, Takumi Okazaki, Hiroshi Imai, “ Implementing the Context Tree Weigghting Method for Text Compression, ” Data Compression Conference (DCC ’00), pp. 123–132, 2000.

[6] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus

[7] 坂口浩章, 川端勉, “ 有限窓を用いた文脈木重みづけ法, ” 電子情報通信学会論文誌 A Vol.J80-A, No.12, pp.2155–2163, 1997.

付録 A　コストの完全性の証明

まず, 準備としてモデルコストに関する再帰的な関係を導く. 文脈木 $\mathcal{T}|s$ が $\{\lambda\}$ でないならば, 文脈木を 0 側と 1 側に分けることができ, ある文脈木 $\mathcal{V}$,$\mathcal{W}$ を用いて

$$\mathcal{T}|s = \mathcal{V} \times 0 \cup \mathcal{W} \times 1 \tag{27}$$

と表すことができる. ここに演算子 $\times$ の意味は

$$\mathcal{V} \times 0 \stackrel{\text{def}}{=} \{s0 | s \in \mathcal{V}\} \tag{28}$$

$$\mathcal{W} \times 1 \stackrel{\text{def}}{=} \{s1 | s \in \mathcal{W}\} \tag{29}$$

である. さらにこのときモデル $\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s} \setminus \{\lambda\}$ はあるモデル $\mathcal{U}_{(0)} \in \mathcal{C}_{\mathcal{V}}$, $\mathcal{U}_{(1)} \in \mathcal{C}_{\mathcal{W}}$ によって

$$\mathcal{U} = \mathcal{U}_{(0)} \times 0 \cup \mathcal{U}_{(1)} \times 1 \tag{30}$$

と書ける. このとき

$$\Gamma_{\mathcal{T}|s}(\mathcal{U}) = 1 + \Gamma_{\mathcal{V}}(\mathcal{U}_{(0)}) + \Gamma_{\mathcal{W}}(\mathcal{U}_{(1)}) \tag{31}$$

が成り立つ.

さて, 式 (15) を証明する. 以下に (15) を再度記載する. 文脈木 $\mathcal{T}$ とノード $s \in \overline{\mathcal{T}}$ に対して

$$\sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} = 1 \tag{32}$$

となる.

証明. 帰納法を用いて証明を行う.

(I) まず, $s \in \mathcal{T}$ のとき (32) が成り立つことを証明する. $s \in \mathcal{T}$ のとき $\mathcal{T}|s = \{\lambda\}$, $\mathcal{C}_{\mathcal{T}|s} = \{\{\lambda\}\}$ より,

$$\sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} = 2^{-\Gamma_{\{\lambda\}}(\{\lambda\})} \tag{33}$$

$$= 1 \tag{34}$$

よって (32) が成り立つ.

(Ⅱ) 次に，$s \notin \mathcal{T}$ かつ $s \in \overline{\mathcal{T}}$ のとき 2 つの文脈 $0s$，$1s$ に対して (32) が成り立つと仮定したとき, 文脈 s に対して (32) が成り立つことを示す.

$$\sum_{\mathcal{U}\in\mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} = 2^{-\Gamma_{\{\lambda\}}(\{\lambda\})} + \sum_{\mathcal{U}\in\mathcal{C}_{\mathcal{T}|s}\setminus\lambda} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \tag{35}$$

$$= \frac{1}{2} + \sum_{\mathcal{U}_{(0)}\in\mathcal{C}_{\mathcal{V}}} \sum_{\mathcal{U}_{(1)}\in\mathcal{C}_{\mathcal{W}}} 2^{-1-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})} \tag{36}$$

$$= \frac{1}{2} + \frac{1}{2}\left(\sum_{\mathcal{U}_{(0)}\in\mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})}\right)\left(\sum_{\mathcal{U}_{(1)}\in\mathcal{C}_{\mathcal{W}}} 2^{-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})}\right) \tag{37}$$

$$= \frac{1}{2} + \frac{1}{2} \tag{38}$$

$$= 1 \tag{39}$$

ここで，(36) は (31) を用いた．(38) は仮定より $\sum_{\mathcal{U}_{(0)}\in\mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})}$，$\sum_{\mathcal{U}_{(1)}\in\mathcal{C}_{\mathcal{W}}} 2^{-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})}$ は 1 であることを用いた．よって (32) が成り立つことを示せた． □

付録 B　無限深さ文脈木重み付け法における重み付け確率の再帰的計算式の導出

定義式 (16) から以下が導出できる．文脈木 $\mathcal{T}$ とノード $s \in \overline{\mathcal{T}}$ に対して

$$P_w^s = \begin{cases} \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}P_w^{1s} & (s \notin \mathcal{T}) \\ P_e(a_s, b_s) & (s \in \mathcal{T}) \end{cases} \tag{40}$$

となる.

証明. (I) で $\mathcal{T} = \{\lambda\}$，(Ⅱ) でその他の場合を証明する.

(I) まず，$\mathcal{T} = \{\lambda\}$ のとき (40) が成り立つことを示す．このとき $\overline{\mathcal{T}} = \{\lambda\}$ なので $s = \lambda$ 以外にはない．一方,$\mathcal{C}_{\mathcal{T}} = \{\{\lambda\}\}$ かつ $\Gamma_{\mathcal{T}}(\{\lambda\}) = 0$ なので (16) より

$$P_w^{\lambda} = \sum_{\mathcal{U}\in\mathcal{C}_{\mathcal{T}|\lambda}} 2^{-\Gamma_{\mathcal{T}|\lambda}(\mathcal{U})} \prod_{u\in\mathcal{U}} P_e(a_{u\lambda}, b_{u\lambda}) \tag{41}$$

$$= P_e(a_\lambda, b_\lambda) \tag{42}$$

よって (40) が成り立つ.

(Ⅱ) 次に $\mathcal{T} = \{\lambda\}$ 以外のとき (40) が成り立つことを示す.

(a)$s \in \mathcal{T}$ のとき，$\mathcal{T}|s = \{\lambda\}$，$\mathcal{C}_{\mathcal{T}|s} = \{\{\lambda\}\}$ より

$$P_w^s = \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{us}, b_{us}) \tag{43}$$

$$= 2^{-\Gamma_{\{\lambda\}}(\{\lambda\})} \prod_{u \in \{\lambda\}} P_e(a_{us}, b_{us}) \tag{44}$$

$$= P_e(a_s, b_s) \tag{45}$$

ただし $\Gamma_{\{\lambda\}}(\{\lambda\}) = 0$ を用いた.

(b)$s \notin \mathcal{T}$ なる $s \in \overline{\mathcal{T}}$ に対して,

$$P_w^s = \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{us}, b_{us}) \tag{46}$$

$$= 2^{-\Gamma_{\mathcal{T}|s}(\{\lambda\})} \prod_{u \in \{\lambda\}} P_e(a_{us}, b_{us}) + \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|s} \setminus \{\lambda\}} 2^{-\Gamma_{\mathcal{T}|s}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{us}, b_{us}) \tag{47}$$

$$= \frac{1}{2} P_e(a_s, b_s) + \sum_{\mathcal{U}_{(0)} \in \mathcal{C}_{\mathcal{V}}} \sum_{\mathcal{U}_{(1)} \in \mathcal{C}_{\mathcal{W}}} 2^{-1-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})} \prod_{u \in \mathcal{U}_{(0)} \times 0 \cup \mathcal{U}_{(1)} \times 1} P_e(a_{us}, b_{us}) \tag{48}$$

$$= \frac{1}{2} P_e(a_s, b_s)$$

$$+ \frac{1}{2} \sum_{\mathcal{U}_{(0)} \in \mathcal{C}_{\mathcal{V}}} \sum_{\mathcal{U}_{(1)} \in \mathcal{C}_{\mathcal{W}}} \left(2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})} \prod_{u \in \mathcal{U}_{(0)} \times 0} P_e(a_{us}, b_{us}) \right) \left(2^{-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})} \prod_{u \in \mathcal{U}_{(1)} \times 1} P_e(a_{us}, b_{us}) \right) \tag{49}$$

$$= \frac{1}{2} P_e(a_s, b_s)$$

$$+ \frac{1}{2} \left(\sum_{\mathcal{U}_{(0)} \in \mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})} \prod_{u \in \mathcal{U}_{(0)} \times 0} P_e(a_{us}, b_{us}) \right) \left(\sum_{\mathcal{U}_{(1)} \in \mathcal{C}_{\mathcal{W}}} 2^{-\Gamma_{\mathcal{W}}(\mathcal{U}_{(1)})} \prod_{u \in \mathcal{U}_{(1)} \times 1} P_e(a_{us}, b_{us}) \right) \tag{50}$$

となる. (48) は (30) を用いた. ここで，$\mathcal{T}|s = \mathcal{V} \times 0 \cup \mathcal{W} \times 1$ のとき,

$$\mathcal{V} = \mathcal{T}|0s = \{r | r0s \in \mathcal{T}\} \tag{51}$$

$$\mathcal{W} = \mathcal{T}|1s = \{r | r1s \in \mathcal{T}\} \tag{52}$$

に注意すると，$0s \in \overline{\mathcal{T}}$ より P_W^{0s} の定義から，

$$\sum_{\mathcal{U}_{(0)} \in \mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(0)})} \prod_{u \in \mathcal{U}_{(0)} \times 0} P_e(a_{us}, b_{us}) \tag{53}$$

$$= \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{u0s}, b_{u0s}) \tag{54}$$

$$= \sum_{\mathcal{U} \in \mathcal{C}_{\mathcal{T}|0s}} 2^{-\Gamma_{\mathcal{T}|0s}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e(a_{u0s}, b_{u0s}) \tag{55}$$

$$= P_W^{0s} \tag{56}$$

が成り立つ．同様に $\sum_{\mathcal{U}_{(1)} \in \mathcal{C}_{\mathcal{V}}} 2^{-\Gamma_{\mathcal{V}}(\mathcal{U}_{(1)})} \prod_{u \in \mathcal{U}_{(1)} \times 1} P_e(a_{us}, b_{us}) = P_W^{1s}$ となり，式 (50) に代入すると式 (40) が成り立つ．

□

付録 C　ソースコード

C.1　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つけ，閾値を設定した場合の圧縮率算出プログラム

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


int max_margin;
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 */



#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;                                      /* 予測確立 */
    double pw;                                      /* 重み付け確立 */
    double tpe;                                     /* 仮予測確立 */
    double tpw;                                     /* 仮重み付け確立 */
    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){                             /* もしセグメントを確保できなかったら error を出す */
        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){                 /* セグメントが最新の物なら更新の必要なし */
        return;
    }

    current_position->newer->older = current_position->older;
    if (manage_oldest == current_position){             /* 最古の物が最新になるので manage_oldest を更新 */
        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;                         /* 最新に更新されたので newer は 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){                /* ルートから葉の（ユニークな）セグメントまでたどる */
        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);
    current_position->tpw = current_position->tpe;          /* 葉では予測確率 = 重み付け確率 */

    temporary_tpw = current_position->tpw;                  /* 親のセグメントの計算で使うために保管 */
    if (update == '0'){                 /* 0 でダミーアップデートしたときには使用したセグメントのリストを更新 */
        listupdate(current_position);
    }
```

```
    while (current_position->parent != NULL){               /* ルートへと辿りながらダミーアップデート */
        current_position = current_position->parent;        /* ひとつ親に移る */

        /* 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--){               /* セグメント内の先頭のノードの重み付け確率を計算 */
            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;              /* 計算結果をセグメントに代入 */

        if (update == '0'){             /* 0 でダミーアップデートしたときには使用したセグメントのリストを更新 */
            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ー時保管 */
    int times;                                              /* 計算回数 */


    for (current_index; current_index >= 0 ;current_index --){/* ルートから系列に従って葉に向かい更新してい
く */
        current_position->pe = current_position->tpe;       /* 予測確率を本更新 */
        current_position->pw = current_position->tpw;       /* 重み付け確率を本更新 */

        if (t == '0'){                                      /* 出力されたシンボルに従いカウンタを更新 */
            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) - log10(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);
                }
            }
```

```
            current_position->log_beta = temporary_log_beta;/* 計算した値を代入 */
        } else {
            break;                                          /* 葉のセグメントまで計算したらループを抜ける */
        }
        /* 次（1 つ子供）のセグメントに移る */
        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 ;/*current_index --*/){
        temporary_index = current_position->index;   /* 保存されている index を変更しないためにコピー */
        if (current_position->length == 0 && current_position->index >= 0){ /* 長さ∞のセグメント */
            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); /* 子供のセグメント
に移る */
                }

            } 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);

}

/* 削除したセグメントの親の長さが∞となる */
```

```
void trim_1(segment *cardinal_position, segment *delete_position)
{
    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;
}
```

/* 兄弟の片方が削除されて親と統合される */

```
void trim_2(segment *cardinal_position, segment *delete_position, segment *delete_brother)
{
    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;
}
```

/* 兄弟の片方が削除されても親と統合されない */

```
void trim_3(segment *cardinal_position, segment *delete_position)
{
    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;                     /* 最も古いセグメントをリストから抜き取る */
    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（パターン 1）*/
        trim_1(cardinal_position, delete_position);
        subtrahend_of_segment = 1;                          /* パターン 1 ではセグメントの減数は 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;                          /* パターン 2 ではセグメントの減数は 2 */

        } else {
            trim_3(cardinal_position, delete_position);
            subtrahend_of_segment = 1;                      /* パターン 3 ではセグメントの減数は 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;
    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 を一時的に保管 */
            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;
}



int check_margin(void)
{
    int i;
    int x = 0;

    for (i = 0; i < LENGTH; i++){
        if (size[i] == 0){
            x++;
        } else {
            break;
        }
    }

    return(x);
}



int main(int argc, char **argv)
{


    FILE *fp;
    int c;
    int j = 0;
    int mask;
    int current_index;                                      /* シンボルの出力個数 */
    int number_of_needs;
    double codeword_length = 0.0;
    int min_leaf_length;
    int k;
    int l = 100;


    if (argc != 3) goto ERROR;
    if (argv[1] == NULL) goto ERROR;
    if (argv[2] == NULL) goto ERROR;
    if ((max_segments = (int)atof(argv[1])) == 0.0) goto ERROR;
    if ((max_margin = (int)atof(argv[2])) == 0.0) goto ERROR;
```

```
if ((fp = fopen("progc.dat", "rb")) == NULL){
    printf("fileopn error\n");
    exit(1);
}

number_of_segments = 0;

root = alloc_segment();                                 /* ルートのセグメントを作る */

manage_newest = root;                                   /* ルートをセグメント更新順リストに追加 */
manage_oldest = root;

current_index = 0;                                      /* 最初はシンボルがないので'0' */
sequence[current_index] = 'e';
while ((c = getc(fp)) != EOF){
    for (mask = 1 << 7; mask != 0; mask >>= 1){

        memset(size, 0, sizeof(int) * LENGTH);          /* 配列 size の初期化 */

        count_size(root);

        min_leaf_length = check_margin();

        if (min_leaf_length > max_margin){
            trim_index(root, min_leaf_length - max_margin);
            trim_sequence(min_leaf_length - max_margin, current_index);
            current_index -= min_leaf_length - max_margin;
        }

        dummyupdate('0');                               /* ダミーアップデート */

        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] = (c & mask)? '1': '0'; /* 出現したシンボルを配列に入れる */
        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]);   /* カウンタなどの更新をする */

        codeword_length = codeword_length - log10(root->pw) / log10(2.0);   /* 符号化後のサイズ */
        j++;

        number_of_needs = trim_confirm(current_index);          /* 新規に必要なセグメント数を確認する */

        while (number_of_segments + number_of_needs > max_segments){
            number_of_segments -= trim_branches();      /* セグメントを 1 つ刈取る */
            number_of_needs = trim_confirm(current_index);
            /* 刈取ったことで新規に必要なセグメント数が変わることがある */
        }

        tree_update(current_index, root);               /* 木の更新（セグメントが増える）*/

    }
}


codeword_length = 8.0 * codeword_length / j;            /*ビットレートを計算*/
printf("%f\n", codeword_length);

fclose(fp);
return(0);
```

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