信州大学工学部

学士論文

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

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　08T2073K
氏名　林 稔章

2012 年 3 月 3 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．従って，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．本研究ではそのデータ圧縮方法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の一つである文脈木重み付け法 (Context-Tree Weighting, CTW) 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 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，木情報源を計算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．更に，有限深さ文脈木重み付け法に対する提案を無限深さ文脈木重み付け法へ適用する方法 [4] も三澤により提案されている．本研究では，[3] で提案された方法を基に更に効率よくメモリを使用する方法を提案し，それぞれの方法で圧縮率を比較することで性能を評価する．その結果，本研究で提案した方法では [3] で提案された方法より圧縮率を向上させることができた．

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 章では効率よくメモリを使用する方法の従来法 [3] と本研究での提案法について説明する．4 章では検証実験方法，結果と考察を示す．5 章でまとめをする．

2　文脈木重み付け法

2.1　木情報源

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．本稿では 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．また，一般的に文字列の語尾を接尾辞 (suffix) と言う．例えば 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．$s \in \mathcal{S}$ の長さ $|s|$ は $|s| \le D$ であると仮定し，この D をモデルの深さと呼ぶ．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．従って，パラメータ全体はパラメータベクトル $\Theta_s \stackrel{\text{def}}{=} \{\theta_s : s \in \mathcal{S}\}$ として表せる．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 に従って x_{n+1} の値が決まる．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 $\mathcal{S}$ を情報源のモデルと呼ぶ．情報源から発生する系列を $x_1x_2\cdots x_t\cdots$ とした時，時刻 t における文脈とは $x_{t-d}^{t-1}(0 \le d \le D)$ である．

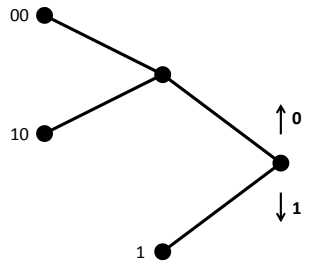

図 1　モデルと完全木

2.2 有限深さ文脈木重み付け法

有限深さ文脈木重み付け法とは，木情報源をモデル，パラメータ両方が未知という条件で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に符号化する確率推定法である．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．有限深さ文脈木重み付け法の最適性は 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あり，各モデルに対して 2.3 節で説明するパラメータ推定を行い 2.4 節で説明するモデルのコストを用いてモデル全体の重み付けで混合することで 2.5 節で示すように系列の出現確率を推定することが出来る．

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 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．

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

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．

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

2.5 重み付け確率

全ての葉が深さ 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

$$P_e^s(x_1^t) \stackrel{\text{def}}{=} P_e(a_s(x_1^t), b_s(x_1^t)) \tag{7}$$

$$P_e^s(x_t|x_1^{t-1}) \stackrel{\text{def}}{=} \begin{cases} \dfrac{a_s(x_1^{t-1}) + \frac{1}{2}}{a_s(x_1^{t-1}) + b_s(x_1^{t-1}) + 1} & (x_t = 0) \\ \dfrac{b_s(x_1^{t-1}) + \frac{1}{2}}{a_s(x_1^{t-1}) + b_s(x_1^{t-1}) + 1} & (x_t = 1) \end{cases} \tag{8}$$

と定めると，式 (3)(4) より次の式が成り立つ．

$$P_e^s(x_1^t) = P_e^s(x_t|x_1^{t-1}) \cdot P_e^s(x_1^{t-1}) \tag{9}$$

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．

$$P_w^s(x_1^t) \stackrel{\text{def}}{=} \sum_{\mathcal{U} \in \mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} \prod_{u \in \mathcal{U}} P_e^{us}(x_1^t) \tag{10}$$

$$\text{このとき} \sum_{\mathcal{U} \in \mathcal{C}_{D-d}} 2^{-\Gamma_{D-d}(\mathcal{U})} = 1 \tag{11}$$

式 (11) はいわゆる等号が成立しているクラフトの不等式 [5] 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 (10) 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 (10) の定義通りの値を再帰的に求めることが出来る．

$$P_w^s(x_1^t) = \begin{cases} \frac{1}{2} P_e^s(x_1^t) + \frac{1}{2} P_w^{0s}(x_1^t) P_w^{1s}(x_1^t) & (s \text{ が文脈木の内部ノード}) \\ P_e^s(x_1^t) & (s \text{ が文脈木の葉}) \end{cases} \tag{12}$$

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 [6] 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 (12) を用いる．

式 (12) 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 [6] 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．

$$\beta_s(x_1^t) \stackrel{\text{def}}{=} \frac{P_e^s(x_1^t)}{P_w^{0s}(x_1^t)P_w^{1s}(x_1^t)} \tag{13}$$

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．

$$P_w^s(x_t|x_1^{t-1}) \stackrel{\text{def}}{=} \frac{P_w^s(x_1^t)}{P_w^s(x_1^{t-1})} \tag{14}$$

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

$$P_w^{ks}(x_1^t) = \begin{cases} P_w^{ks}(x_t|x_1^{t-1}) \cdot P_w^{ks}(x_1^{t-1}) & (ks \text{ が } x_{-\infty}^{t-1} \text{の接尾辞}) \\ P_w^{ks}(x_1^{t-1}) & (ks \text{ が } x_{-\infty}^{t-1} \text{の接尾辞でない}) \end{cases} \tag{15}$$

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 (12) は以下のように表せる．

$$P_w^s(x_1^t) = \frac{1}{2}P_e^s(x_1^t) + \frac{1}{2}P_w^{ks}(x_t|x_1^{t-1}) \cdot P_w^{0s}(x_1^{t-1}) \cdot P_w^{1s}(x_1^{t-1}) \tag{16}$$

ここで，β_s と式 (9)(16) を用いると式 (14) は以下のように表すことができる．

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

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

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

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 (12) 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 (12)(17) を用いて以下のように計算する．

$$P_w^s(0|x_1^{t-1}) = \begin{cases} \dfrac{P_e^s(0|x_1^{t-1}) \cdot \beta_s(x_1^{t-1}) + P_w^{ks}(0|x_1^{t-1})}{\beta_s(x_1^{t-1}) + 1} & (s \text{ が文脈木の内部ノード}) \\ P_e^s(0|x_1^{t-1}) & (s \text{ が文脈木の葉}) \end{cases} \tag{19}$$

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 (18) を用いて以下のように更新し，カウンタ a_s を 1 増やす．

$$\beta_s(x_1^t) = \frac{P_e^s(0|x_1^{t-1})}{P_w^{ks}(0|x_1^{t-1})} \beta_s(x_1^{t-1}) \tag{20}$$

5. $x_t = 1$ だった場合，以下のように式 (19) を計算し直す．

$$P_w^s(1|x_1^{t-1}) = \begin{cases} \dfrac{P_e^s(1|x_1^{t-1}) \cdot \beta_s(x_1^{t-1}) + P_w^{ks}(1|x_1^{t-1})}{\beta_s(x_1^{t-1}) + 1} & (s \text{ が文脈木の内部ノード}) \\ P_e^s(1|x_1^{t-1}) & (s \text{ が文脈木の葉}) \end{cases} \tag{21}$$

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．

$$\beta_s(x_1^t) = \frac{P_e^s(1|x_1^{t-1})}{P_w^{ks}(1|x_1^{t-1})} \beta_s(x_1^{t-1}) \tag{22}$$

以上の手順 1〜5 を次のシンボルが無くなるまで行う．

3 メモリの使用効率を向上させる方法

3.1 メモリの使用効率を上げるための従来法

有限深さ文脈木重み付け法は文脈木のノード数により使用するメモリ数が変わるため文脈木の深さ D により符号化に必要なメモリ数が決まる．また，D を大きくするとより長い文脈での重み付け確率を算出出来るため圧縮率が良くなる傾向がある．しかし，D を大きくすることにより出現しない文脈が多くなり符号化に使用しない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メモリを持つノード数の上限を設定し，符号化を開始する．そして，ノードを使用する際にメモリを確保する．符号化中にメモリを持つノード数が上限に達した時には，メモリを持つノードの中で更新が一番古いノードのメモリを消去する．これは，更新が古いノードに対応する文脈はメモリを割り当てたノードに対応する文脈の中で一番出現しない文脈であり，圧縮率への影響が少ないと考えられるためである．ノードからメモリを削除しても，ノードに対応する文脈が出現しなかったことになるだけであり重み付け確率の計算方法には問題は生じない．なお，ノードのメモリを消去した際には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．この時，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，こ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はないからである．また，この [3] で提案された方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．この時，D より深いノードは生成できないため文脈木の深さ D はできるだけ大きくすると良い．

図 2 に $x_{1-D}^0 = \cdots 00, x_1^t = 11$ 深さ $D = 2$，メモリ上限 5 の例を示す．図 2 の黒丸はメモリ，実線はノードにメモリ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．また，①〜⑤の数字はメモリを持つノードの新しさを表して

おり数字が小さい程新しいノードである．①〜⑤のノードを使用する際にはまだ上限に達していないためメモリを確保することが出来る．しかし，文脈 11 に対応するノードにメモリを割り当てる際にメモリ数の上限を超えてしまう為，最も前に使用したノードのメモリである⑤のメモリを削除している．

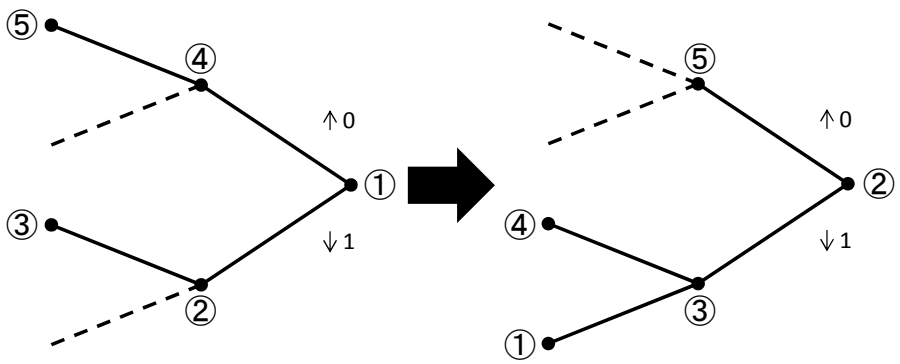

図 2　メモリの削除方法 (メモリ上限 5)

このようにメモリを持つノードの中で最も前に使用したノードのメモリを削除することでメモリ数の上限を超えないようにすることが出来る．この方法を用いれば D を大きくすることで従来法より深いノードのメモリを確保することが出来，無駄になっていたメモリを有効に利用できる．

3.2　メモリの使用効率を上げるための提案法

[3] で提案された方法では出現した文脈に対応するノードには必ずメモリを割り当てており，その後しばらくその文脈が出現しないとしても‘一番古いノード’になるまでメモリを保有し続ける．その間，このノードは符号化確率の推定に利用されず，より頻繁に出現する他の文脈からメモリを奪っていることになる．そこで，[3] で提案された方法ではメモリを持つノードのみの新しさでメモリを消去するかどうかを考えていたが，本研究での提案ではメモリを持つ持たないに関わらず全てのノードの新しさを考える．そして，符号化でメモリが無いノードが参照された際に，そのノードより古いノードの中にメモリを持つノードがあればその古いノードのメモリを削除し符号化で参照されたノードにメモリを付け替える．そうではなく，符号化で参照されたノードより古いノードの中にメモリを持つノードが無い場合にはメモリを持たないまま‘新しいノード’になる．参照されたノードより古いノードの中にメモリを持つノードが複数ある場合にはその中で一番古いノードのメモリを付け替える．また，メモリを持たないまま‘新しいノード’になった時はカウンタ $(a, b) = (0, 0)$ として重み付け確率を計算する．

この提案法を用いることにより殆ど使用されないノードはその文脈が出現してもメモリを割り当てられず，そのメモリをより頻繁に出現するノードに割り当てることができる．つまり，圧縮率への影響が大きいメモリに優先的にメモリを割り当てることができるため [3] で提案された方法よりも圧縮率を向上させることができると期待できる．

3.3 提案法の実装

図 3 に $x_{1-D}^0 = \cdots 00, x_1^t = 11$ 深さ $D = 2$，メモリ上限 5 とした時の本研究での提案法でメモリの削除方法を示す．図 2 の時と同様に黒丸はメモリ，実線はノードにメモリ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．まず，文脈木の根 (λ) が一番新しいノードになるようにノードの新しさを考える．これは，λ が最も重要な文脈であるためである．ノードの新しさは①〜⑦で表し，数字が小さい程新しいノードである．割り当てられたメモリの数が上限に達するまでは [3] で提案された方法と同じようにメモリを割り当てる．そして，文脈 11 に対応するノードが参照された時にはメモリ数は上限に達している．この時，[3] で提案された方法ではメモリを持つノードの中で一番古いノードを削除していたが，この方法では 11 のノードより古いノードは無いため，メモリを持たないまま‘新しいノード’になっている．もし参照されたノードより古いノードの中にメモリを持つノードがあった場合にはそのメモリを削除し参照されたノードにメモリを付け替える。このように，全てのノードで新しさを考えることでメモリの上限を超えないまま [3] の方法より重要な文脈にメモリを割り当てることが出来る．

4 検証

4.1 検証の方法

2.6 節の文脈木の全てのノードがメモリを持つ方法，3.1 節のメモリの使用効率を向上させる方法の従来法 [3]，そして 3.2 節で述べた本研究での提案法のそれぞれの方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化を行いメモリの上限を変化させて圧縮率を算出する．本稿では圧縮率は [3][4] 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はカルガリーコーパス [7] 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．また，[3] でも圧縮率が算出されているが [3] では実際に算術符号化を行なっておらず，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符号語長に基づいて圧縮率を算出している．

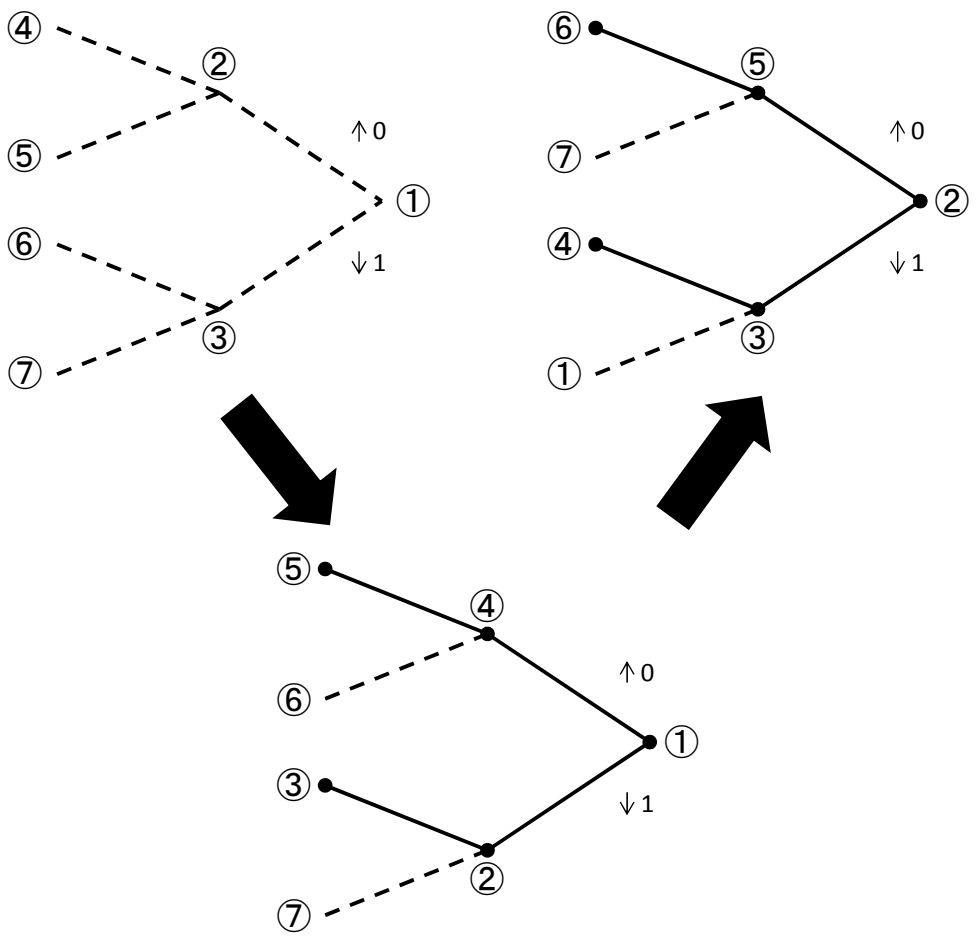

図 3　本研究の提案法でのメモリの削除方法 (メモリ上限 5)

4.2　結果

それぞれの方法でノードに割り当てるメモリ数を変化させ，カルガリーコーパスを圧縮した結果を図 4〜21 に示した．文脈木の全てのノードがメモリを持つ方法を original method，[3] で提案された方法を conventional method，そして本研究での提案法を proposed method とした．なお，original method では全てのノードがメモリを持つため文脈木の深さ D を変化させ，文脈木の深さ D のノード数である $2^{D+1}-1$ をノードに割り当てるメモリの数とした．また，他の 2 つの提案法では木の深さ D を設定する必要があるため，この実験では $D=25$ として検証を行った．

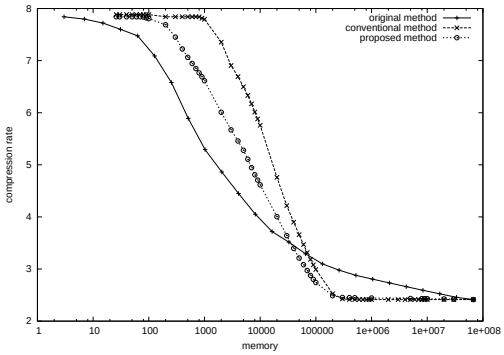

図 4　従来法と提案法の圧縮率比較 (bib)

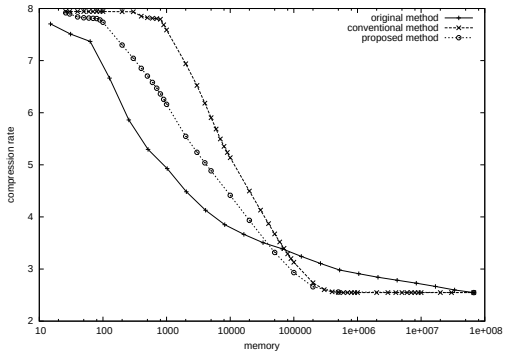

図 5　従来法と提案法の圧縮率比較 (book1)

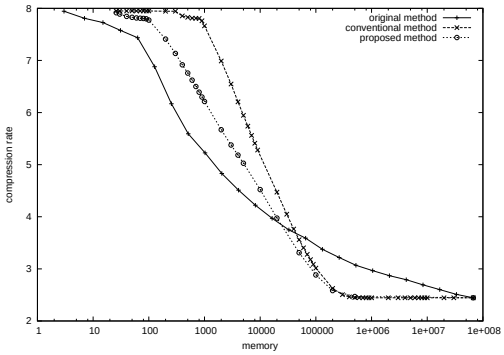

図 6　従来法と提案法の圧縮率比較 (book2)

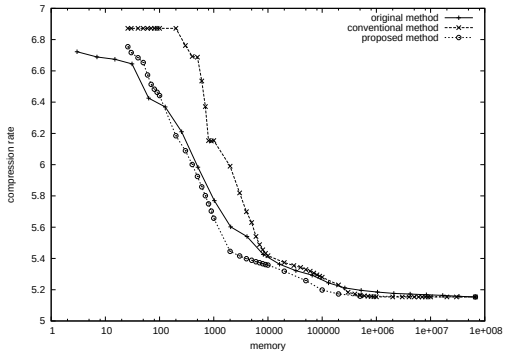

図 7　従来法と提案法の圧縮率比較 (geo)

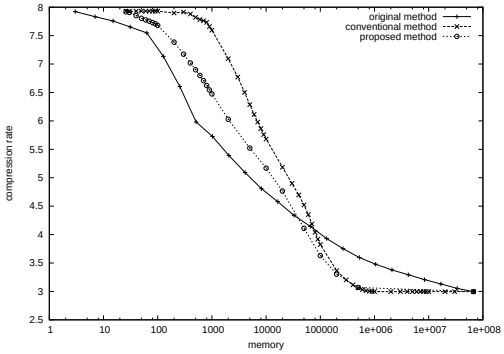

図 8　従来法と提案法の圧縮率比較 (news)

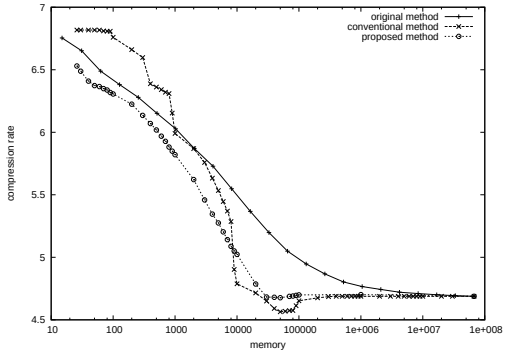

図 9　従来法と提案法の圧縮率比較 (obj1)

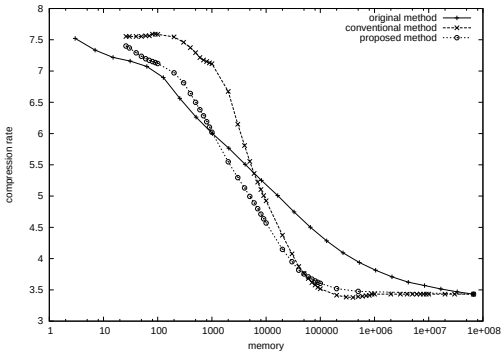

図 10　従来法と提案法の圧縮率比較 (obj2)

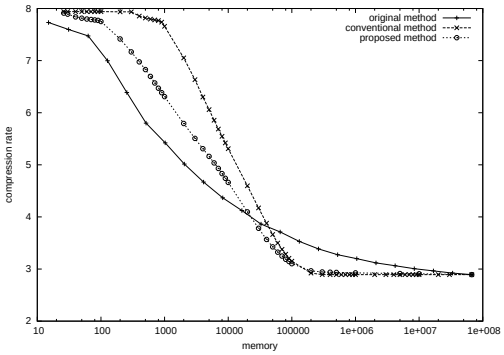

図 11　従来法と提案法の圧縮率比較 (paper1)

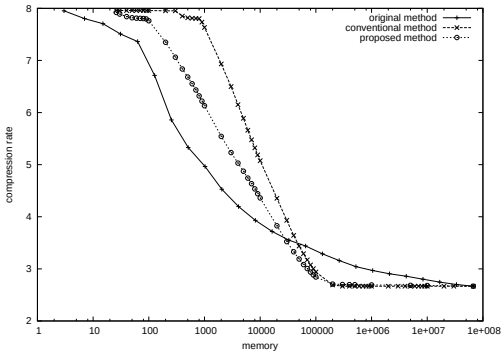

図 12　従来法と提案法の圧縮率比較 (paper2)

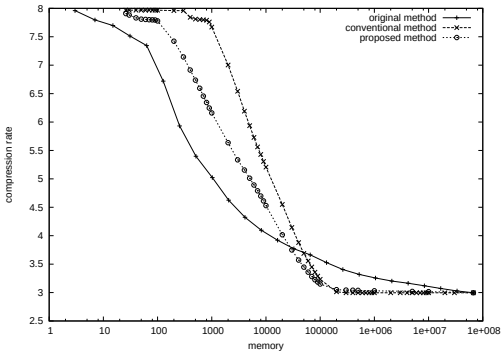

図 13　従来法と提案法の圧縮率比較 (paper3)

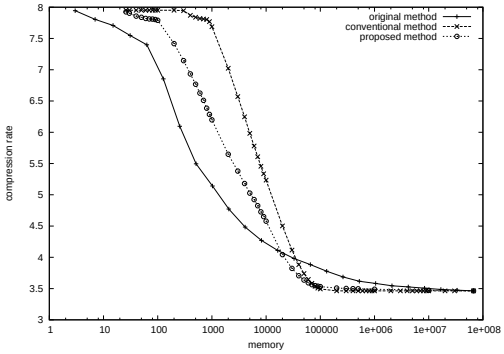

図 14　従来法と提案法の圧縮率比較 (paper4)

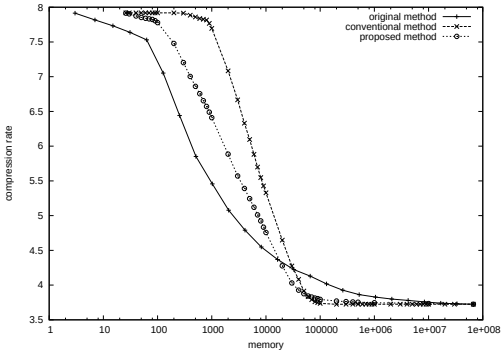

図 15　従来法と提案法の圧縮率比較 (paper5)

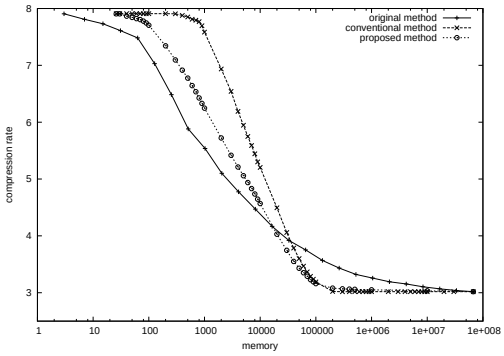

図 16　従来法と提案法の圧縮率比較 (paper6)

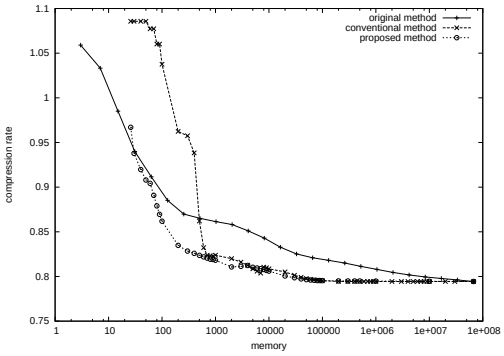

図 17　従来法と提案法の圧縮率比較 (pic)

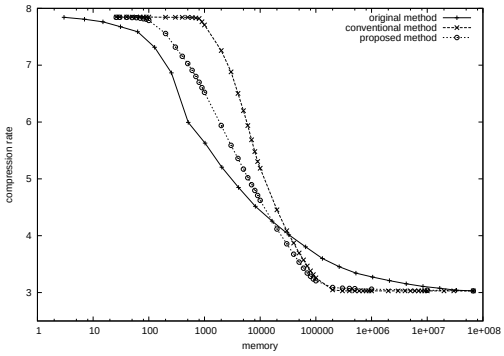

図 18　従来法と提案法の圧縮率比較 (progc)

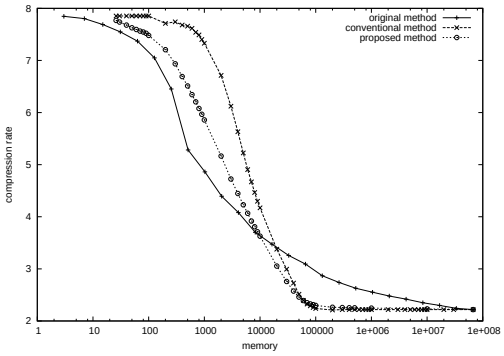

図 19　従来法と提案法の圧縮率比較 (progl)

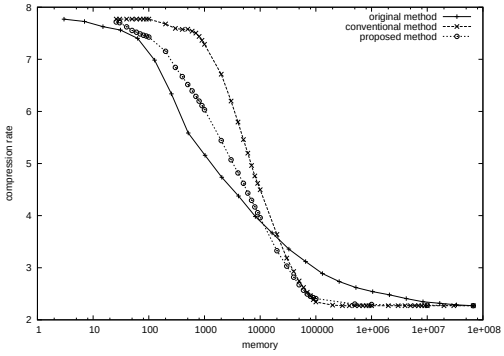

図 20　従来法と提案法の圧縮率比較 (progp)

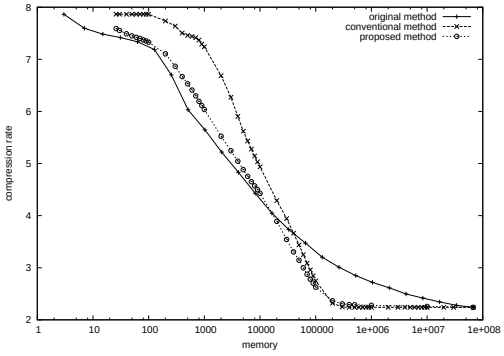

図 21　従来法と提案法の圧縮率比較 (trans)

4.3 考察

まず，従来法 (original method) のみに着目すると図 4〜6，8，11〜16，18〜21 のテキストファイルについては深さ $D = 6$，つまりメモリの上限が 100 のあたりで圧縮率が一気に良くなっていることが分かる．これは，[3] にも述べられているように CTW 法はバイナリの文脈木を用いており，テキストの一文字を 8[bit] に分解して符号化を行なっているため 6[bit] 見るとその文字が絞り込まれるためであると考えられる．従って図 7，9，10，17 のバイナリファイルではこのような特徴は見られない．

次に，従来法と 2 つの提案法 (conventional method, proposed method) を比べると全てのファイルでメモリ数が少ない場合には従来法の方が圧縮率が良く，メモリ数を多くすると他の 2 つの方法の方が圧縮率が良くなっていることが分かる．これも [3] で述べられているが，メモリ数が少ない時には従来法ではほぼ全てのメモリを使用しており無駄が少なく，メモリが削除されないためである．提案法ではメモリを削除しているためその分圧縮率が悪くなっていると考えられる．一方，メモリ数が多い場合従来法では使用していないメモリが多くなり他の 2 つの提案法ではその使用してないメモリも符号化に利用しているため圧縮率が良くなっている．また，メモリ数が約 10^5 以上の場合で従来法での最良 ($D = 25$) の圧縮率と等しくなる．これは，[3] で提案された方法と本研究の提案法では $D = 25$ の文脈木を考えており，$D = 25$ の文脈木を用いて従来法によって圧縮する際に使用する全てのノードにメモリが確保されたためである．つまり，従来法での $D = 25$ の場合と同様の圧縮を少ないメモリ数で実現している．従って，それ以上メモリ数を増加させても使用しないメモリを増やすだけで圧縮率は変わっていない．

最後に [3] で提案された方法と本研究での提案法を比べると，圧縮率が下限に達していないメモリ数約 10^5 未満では本研究での提案法の方が図 9，10 を除く全てのファイルで圧縮率が良くなっている．これは，本研究での提案では [3] で提案された方法では削除されていた頻繁に出てくる重要なノードのメモリを確保できるためであると考えられる．しかしながら，本研究での提案法の方が [3] で提案された方法より少ないメモリ数で圧縮率が従来法での $D = 25$ の場合と等しくなっているわけではなく，ほぼ同じメモリ数で圧縮率の下限に達している．これは，先ほど述べたようにノードに割り当てるメモリの数が従来法での $D = 25$ の文脈木を用いて圧縮する時に使用する全てのノードにメモリが確保できるメモリ数と一致するメモリ数で圧縮率は下限に達するためである．従来法と圧縮率が一致するには使用するノードの全てにメモリが必要であり，その使用されるノード全てにメモリが確保できる数で 2 つの提案法の圧縮率は下限に達している．

提案法で文脈木の深さ D を大きくしていくとより長い文脈を考えることが出来るため圧縮率は全体的に良くなると考えることが出来る．しかしながら，D が大きいほど D を大き

くしても圧縮率の変化は少なくなる．例えば図 9(obj1) の従来法をみると $D = 20$(メモリ数約 2×10^6) と $D = 25$(メモリ数約 7×10^7) では圧縮率は 約 0.055[bit/シンボル] しか変化が無い．提案法の圧縮率の上限は設定した D の従来法での圧縮率と一致するため，提案法で D を 25 より大きく設定しても圧縮率は僅かに良くなるがあまり変わらないと予想することが出来る．このことから D の値は，従来法での圧縮率の変化がほとんど無くなる D の値を設定すれば良いと考えられる．

以上のことから本研究での提案を用いることにより [3] で提案された方法より更に効率よくメモリを使用することが出来ることがわかった．しかし，本研究での提案法ではメモリを付け替える際に参照されたノードより古いノードにメモリがあるかどうかを探す必要があり，この作業に時間がかかるため他の方法よりプログラムの実行時間が長くなるというデメリットが存在する．

5 まとめ

[3] で提案された方法を参考に有限深さ文脈木重み付け法で効率よくメモリを使用する方法を提案し，提案法の性能評価を行った．提案法とは，全てのノードの新しさを考え設定したメモリ数の上限に達した際に符号化確率の計算に使用するノードより古いノードの中にメモリを持つノードがあればそのメモリと付け替え，無ければメモリを持たないまま新しいノードにする方法である．この提案法と有限深さ文脈木重み付け法で文脈木の全てのノードにメモリを割り当てる従来法，メモリ数を設定し上限に達した際に古いメモリを持つノードを削除しメモリを付け替える方法 [3] で実際にファイルの圧縮を行い圧縮率を比較した．その結果，[3] で提案された方法より更に圧縮率を向上させることができた．しかしながら，本研究での提案法を用いたプログラムの実行速度は非常に遅いため今後プログラムを改良する必要がある．また，[3] で提案された方法を無限深さ文脈木重み付け法に適用する方法 [4] も提案されているため，その方法を基に本研究での提案を無限深さ文脈木重み付け法へ適用させることができると考えられる．

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

[6] Kunihiko Sadakane, Takumi Okazaki, Hiroshi Imai, “Implementing the Context-Tree Weighting Method for Text Compression,” Data Compression Conference(DCC’00), pp123–132, 2000.

[7] ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus，2011 年 10 月閲覧.

付録 A　ソースコード

A.1　ノードの再帰順位に基づきノードに割り当てるメモリ数に制限をつけた場合での圧縮プログラム

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

#define DEPTH 25

#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;

FILE *outfp;

typedef struct node{
    long a;
    long b;
    double pw;
    double alpha;
} node;

typedef struct newness{
    int newer;
    int older;
    node *nodes;
} newness;
struct node *nodes;
newness branches[(1UL << (DEPTH + 1)) - 1];

int MAX_NODES;
int newest;
int oldest;
int oldestnode;
char history[DEPTH];
int latest = DEPTH;

void printlist()
{
    int n;

    printf("list:");
    for (n = newest; n != -1; n = branches[n].older){
        printf(" %d", n);
    }
    printf("\n");
    return;
}

int findleaf()
{
    int n = 0;
    int i;
```

```
    for (i = 0; i < DEPTH; i++){
        if (history[(latest - i + DEPTH) % DEPTH] == '0') {
            n = 2*n+1;
        } else {
            n = 2*n+2;
            }
        }
    return(n);
}

void oldestnode_update(int n)
{
    int i;

    for (i = branches[n].newer; i != newest; i = branches[i].newer){
        if (branches[i].nodes != NULL){
            oldestnode = i;
            return;
        }
    }
    printf("error in %d\n", __LINE__);
    exit(1);
}

void reuse_node(int n)
{
    int i;
    int j;

    if (branches[n].nodes != NULL){
        printf("error in %d\n", __LINE__);
        exit(1);
    }

    for (i = oldestnode, j = n; i != newest && j != newest ; i = branches[i].newer, j = branches[j].newer){
        if (i == n){
            branches[n].nodes = branches[oldestnode].nodes;
            branches[oldestnode].nodes = NULL;
            branches[branches[n].newer].older =  branches[n].older;
            if (branches[n].older == -1){
                oldest = branches[n].newer;
            } else {
                branches[branches[n].older].newer = branches[n].newer;
            }
            oldestnode_update(oldestnode);
            branches[n].nodes->a = 0;
            branches[n].nodes->b = 0;
            branches[n].nodes->pw = 0;
            branches[n].nodes->alpha = 0.0;

            branches[newest].newer = n;
            branches[n].older = newest;
            branches[n].newer = -1;
            newest = n;
            return;
        }

        if (j == oldestnode){
            branches[branches[n].newer].older =  branches[n].older;
            if (branches[n].older == -1){
                oldest = branches[n].newer;
            } else {
                branches[branches[n].older].newer = branches[n].newer;
            }

            branches[newest].newer = n;
            branches[n].older = newest;
            branches[n].newer = -1;
```

```
            newest = n;
            return;

        }
    }

    if (i == newest){
        branches[branches[n].newer].older =  branches[n].older;
        if (branches[n].older == -1){
            oldest = branches[n].newer;
        } else {
            branches[branches[n].older].newer = branches[n].newer;
        }

        branches[newest].newer = n;
        branches[n].older = newest;
        branches[n].newer = -1;
        newest = n;
        return;
    } else if (j == newest) {
        branches[n].nodes = branches[oldestnode].nodes;
        branches[oldestnode].nodes = NULL;

        branches[n].nodes->a = 0;
        branches[n].nodes->b = 0;
        branches[n].nodes->pw = 0;
        branches[n].nodes->alpha = 0.0;
        branches[branches[n].newer].older =  branches[n].older;
        if (branches[n].older == -1){
            oldest = branches[n].newer;
        } else {
            branches[branches[n].older].newer = branches[n].newer;
        }
        oldestnode_update(oldestnode);
        branches[newest].newer = n;
        branches[n].older = newest;
        branches[n].newer = -1;
        newest = n;
        return;
    } else{
        printf("error in %d\n", __LINE__);
        exit(1);
    }
}

void listupdate(int n)
{
    if (n == oldestnode){
        oldestnode_update(n);
    }
    if (branches[n].newer == -1){
        return;
    } else {
        branches[branches[n].newer].older = branches[n].older;
    }
    if (branches[n].older == -1){
        oldest = branches[n].newer;
    } else {
        branches[branches[n].older].newer = branches[n].newer;
    }
    branches[newest].newer = n;
    branches[n].older = newest;
    branches[n].newer = -1;
    newest = n;
    return;
}

void dummyupdate()
{
```

```
    int n = findleaf();
    int d;
    double prev_pw;

    if (branches[n].nodes == NULL){
        reuse_node(n);
    } else {
        listupdate(n);
    }
    if (branches[n].nodes != NULL){
        prev_pw = (branches[n].nodes->a + 0.5) / (branches[n].nodes->a + branches[n].nodes->b + 1.0);
    } else {
        prev_pw = 0.5;
    }
    for (d = DEPTH - 1, n = (n - 1) / 2; d >= 0; d--, n = (n - 1) / 2){
        if (branches[n].nodes == NULL){
            reuse_node(n);
        } else {
            listupdate(n);
        }
        if (branches[n].nodes != NULL){
            branches[n].nodes->pw = (branches[n].nodes->a + 0.5)
                                  / (branches[n].nodes->a + branches[n].nodes->b + 1.0)
                                  / (1.0 + exp(-branches[n].nodes->alpha)) + prev_pw
                                  / (exp(branches[n].nodes->alpha) + 1.0);
            prev_pw = branches[n].nodes->pw;
        } else {
            prev_pw = 0.5;
        }

        if (((unsigned int *)&(prev_pw))[0] == 0x00000000 && ((unsigned int *)&(prev_pw))[1] == 0xfff80000){
            printf("!!!!%d!!!!\n", __LINE__);
            printf("%g %g\n", branches[n].nodes->alpha, exp(branches[n].nodes->alpha));
            exit(1);
        }
    }
    return;
}

void actualupdate(char t)
{
    int nc = findleaf();
    int np;
    double prev_pw;

    if (t == '0'){
        if (branches[nc].nodes != NULL){
            prev_pw = (branches[nc].nodes->a + 0.5) / (branches[nc].nodes->a + branches[nc].nodes->b + 1.0);
            branches[nc].nodes->a++;
        } else {
            prev_pw = 0.5;
        }
        np = (nc - 1) / 2;
        if (branches[np].nodes != NULL){
            branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->a + 0.5)
                          - log(branches[np].nodes->a + branches[np].nodes->b + 1.0) - log(prev_pw);
            branches[np].nodes->a++;
        }
        for (nc = np, np = (np - 1) / 2; nc > 0; nc = np, np = (np - 1) / 2){
            if (branches[np].nodes != NULL){
                if (branches[nc].nodes != NULL){
                    branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->a + 0.5)
                                              - log(branches[np].nodes->a + branches[np].nodes->b + 1.0)
                                              - log(branches[nc].nodes->pw);
                } else {
                    branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->a + 0.5)
                                            - log(branches[np].nodes->a + branches[np].nodes->b + 1.0) - log(0.5);
                }
                if (branches[np].nodes->alpha > 1000.0){
```

```
                    branches[np].nodes->alpha = 1000.0;
                }
                if (branches[np].nodes->alpha < -700){
                    branches[np].nodes->alpha = -700;
                }
                branches[np].nodes->a++;
            }
        }
    } else {
        if (branches[nc].nodes != NULL){
            prev_pw = (branches[nc].nodes->b + 0.5) / (branches[nc].nodes->a + branches[nc].nodes->b + 1.0);
            branches[nc].nodes->b++;
        } else {
            prev_pw = 0.5;
        }
        np = (nc - 1) / 2;
        if (branches[np].nodes != NULL){
            branches[np].nodes->pw = (branches[np].nodes->b + 0.5)
                                   / (branches[np].nodes->a + branches[np].nodes->b + 1.0)
                                   / (1.0 + exp(-branches[np].nodes->alpha)) + prev_pw
                                   / (exp(branches[np].nodes->alpha) + 1.0);
            branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->b + 0.5)
                                   - log(branches[np].nodes->a + branches[np].nodes->b + 1.0) - log(prev_pw);
            if (branches[np].nodes->alpha > 1000.0){
                branches[np].nodes->alpha = 1000.0;
            }
            if (branches[np].nodes->alpha < -700){
                branches[np].nodes->alpha = -700;
            }
            branches[np].nodes->b++;
            prev_pw = branches[np].nodes->pw;
        } else {
            prev_pw = 0.5;
        }
        for (nc = np, np = (np - 1) / 2; nc > 0; nc = np, np = (np - 1) / 2){
            if (branches[np].nodes != NULL){
                branches[np].nodes->pw = (branches[np].nodes->b + 0.5) / (branches[np].nodes->a
                                       + branches[np].nodes->b + 1.0) / (1.0 + exp(-branches[np].nodes->alpha))
                                       + prev_pw / (exp(branches[np].nodes->alpha) + 1.0);
                prev_pw = branches[np].nodes->pw;
                if (branches[nc].nodes != NULL){
                    branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->b + 0.5)
                                              - log(branches[np].nodes->a + branches[np].nodes->b + 1.0)
                                              - log(branches[nc].nodes->pw);
                } else {
                    branches[np].nodes->alpha = branches[np].nodes->alpha + log(branches[np].nodes->b + 0.5)
                                              - log(branches[np].nodes->a + branches[np].nodes->b + 1.0) - log(0.5);
                }
                if (branches[np].nodes->alpha > 1000.0){
                    branches[np].nodes->alpha = 1000.0;
                }
                if (branches[np].nodes->alpha < -700){
                    branches[np].nodes->alpha = -700;
                }
                branches[np].nodes->b++;
            } else {
                prev_pw = 0.5;
            }
        }
    }
    return;
}

void inithistory(void)
{
    int i;

    for (i = 0; i < DEPTH; i++){
        history[i] = '0';
```

```
    }
    return;
}

void historyupdate(char t)
{
    latest = (latest + 1) % DEPTH;
    history[latest] = t;
    return;
}

void initbranches(void)
{
    int i;

    for (i = 0; i < MAX_NODES ; i++){
        nodes[i].a = 0;
        nodes[i].b = 0;
        nodes[i].pw = 0;
        nodes[i].alpha = 0.0;
    }
    for (i = 0; i < (1UL << (DEPTH + 1)) - 1; i++){
        branches[i].newer = i - 1;
        branches[i].older = i + 1;
    }
    branches[(1UL << (DEPTH + 1)) - 2].older = -1;
    newest = 0;
    oldest = (1UL << (DEPTH + 1)) - 2;
    for (i = 0; i < MAX_NODES; i++){
        branches[i].nodes = nodes + i;
    }
    oldestnode = MAX_NODES - 1;
    for (; i < (1UL << (DEPTH + 1)) - 1; i++){
        branches[i].nodes = NULL;
    }
    return;
}

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

void output_bit(int bit)
{
    putc((bit)? '1': '0', outfp);

    /*buffer >>= 1;
    if (bit) buffer |= 0x80;
    bits_to_go -= 1;
    if (bits_to_go == 0){
        putc(buffer, outfp);
        bits_to_go = 8;
    }*/

    return;
}

void done_outputing_bits()
{
    putc(buffer >> bits_to_go, outfp);
    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)
{
    long range = (long)(high - low) + 1;
    long bound = low + range * branches[0].nodes->pw;

    if (bound == low){
        bound++;
    }
    if (bound == high + 1){
        bound--;
    }
    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;
}

main(int argc, char **argv)
{
    FILE *srcfp;
    int c;
```

```
    int mask;
    char t;

    latest = DEPTH;
    MAX_NODES = atoi(argv[3]);

    nodes = (struct node *)calloc(MAX_NODES,sizeof(struct node));
    if (nodes == NULL){
        printf("calloc() error in %d\n", __LINE__);
        exit(1);
    }

    initbranches();
    inithistory();

    start_outputing_bits();
    start_encoding();

    if ((srcfp = fopen(argv[1], "rb")) == NULL){
        printf("fileopen error on %d\n", __LINE__);
        exit(1);
    }
    if ((outfp = fopen(argv[2], "w")) == NULL){
        printf("fileopen error on %d\n", __LINE__);
        exit(1);
    }

    while ((c = getc(srcfp)) !=EOF){
        for (mask = 1 << 7; mask != 0; mask >>= 1){
            t = (c & mask)? '1': '0';
            dummyupdate();
            encode_symbol(t);
            actualupdate(t);
            historyupdate(t);
        }
    }
    done_encoding();
    //done_outputing_bits();

    fclose(srcfp);
    fclose(outfp);
    return(0);
}
```