信州大学工学部

学士論文

有限メモリで動作する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　05T2095K
氏名　三澤 陽一

2009 年 3 月 22 日

目次

1 序章

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文脈木重み付け（CTW）法 [1] がある．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．一般的にデータを圧縮するには, 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．そのため一度情報源系列を全て見て, その後符号化を開始する必要がある．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, 情報源を調べながら符号化を行うことができる．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．しかし, この符号化を継続的に行うには限りなくメモリが必要である．そこで, 本研究ではデータ圧縮の手法の一つである文脈木重み付け法を有限メモリで実現する方法について考えた．そのために, 使用可能メモリを制限する方法を提案する．また, この方法の圧縮率を圧縮し性能を評価する．

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 CTW 法について説明する．3 章では圧縮率を算出するにあたっての改良点の説明を行う．4 章では提案法についての説明と, 結果と評価を示す．5 章ではまとめをする．

2 文脈木重み付け法について

2.1 木情報とは

木情報源とは文脈木で定まる Markov 情報源の一種である．木情報源はアルファベットを {0,1} と仮定すると系列 $x_{-\infty}^{\infty}$ を生み出す．x_m^n は系列 x_m,x_{m+1},$\cdots$,x_n を意味している．木情報源の確率的なふるまいは完全な接尾辞の集合 $\mathcal{S}$ で表わすことができる．$s \in \mathcal{S}$ の長さは D 以下とする．この D は文脈木の深さと呼ばれる．$\mathcal{S}$ の各々の接尾辞 s にパラメータ θ_s が存在する．各パラメータは [0,1] の値となり {0,1} の分布を指す．このように全てのパラメータはパラメータベクトル $\Theta_{\mathcal{S}} \stackrel{\text{def}}{=} \{\theta_s : s \in \mathcal{S}\}$ を作る．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 は唯一に決まる．この s を x_m^n の文脈という．このとき s に対応する θ_s が決まり, θ_s にしたがいシンボルが生み出される．

2.2 文脈木重み付け法とは

2 値の無記憶情報源において ‘1’ の出る確率を θ とすると, 系列に ‘0’ が a 個,‘1’ が b 個含まれている情報源系列 $x_1^N = x_1 x_2 \cdots x_N$ の出現確率は,

$$P(x_1 x_2 \cdots x_N) = (1-\theta)^a \theta^b \tag{1}$$

と表せる．ここから θ の事前分布として Jeffreys 事前分布 $\pi^1 \theta^{-1/2}(1-\theta)^{-1/2}$ を用いると,Krichevsky-Trofimov(KT) 推定を得ることができる．KT 推定確率 $P_e(a,b)$ は $P_e(0,0) = 1$ として次のように逐次的に得ることができる．

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

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

次に木情報源から発生する文字列を圧縮することを考える．このときエンコーダとデコーダには情報源全体の確率分布は知られていない．この状況で符号化分布重み付けを行う．まず, 深さが D の文脈木 $\mathcal{T}_D$ を用意する. 各ノードは ‘0’ と ‘1’ に分かれている．ある文脈 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 $s \in \mathcal{T}_D$ には $a_s \geq 0$ と $b_s \geq 0$ が対応する．親ノード s の子供 $0s$ と $1s$ は, カウント $a_{0s} + a_{1s} = a_s$ と $b_{0s} + b_{1s} = b_s$ を満たす.

今, 各ノードに重み付け確率が割り振られている. この重み付け確率は, 文脈木 $\mathcal{T}_D$ 上で再帰的に定められる. ノード s に割り振られている重み付け確率 P_w^s は, 以下のように計算できる.

$$P_w^s = \begin{cases} \frac{1}{2}P_e(a_s, b_s) + \frac{1}{2}P_w^{0s}P_w^{1s} & (0 \leq l(s) < D) \\ P_e(a_s, b_s) & (l(s) = D) \end{cases} \tag{4}$$

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 における文脈を x_{t-d}^{t-1},$(0 \leq d \leq D)$ とする. また, $x_t^{t-1} = \lambda$ と定義する. この λ は空系列を意味している.

以下に重み付け確率算出の手順を示す.

1. まず文脈 x_{t-D}^{t-1} 読む. その文脈に沿い文脈木を λ から文脈に対応した葉までたどる.

2. 葉から λ まで文脈に沿ってさかのぼりながら P_e, P_w を計算していく. これらは式 (2),(3),(4) で定義したとおり, 葉から再帰的に計算される. ここで, p_wを計算する際に, 文脈木の葉においては子供のノードがないため, $P_w = P_e$となっている.なお, エンコーダはx_1^t を知らないため, x_1^t が'0' であるとして計算を行う.

3. λ まで終わったら, そのときの P_w^λ を系列 x_1^t の符号化確率とする.

4. その後情報源からシンボルが発生し '1' だった場合は '1' として計算しなおし, 次のシンボルの符号化確率の計算に入る.

また選択確率を $\frac{1}{2}$ とせずに γ,$1-\gamma$ $(0 \leq \gamma \leq 1)$ を用いることも可能である.

Willems らは, この系列の符号化確率から逐次的な符号化確率分布を調べることで算術符号により情報源を符号化することが可能であることを述べている.

2.3 文脈木重み付け法の符号化確率の計算例

文脈木を $\mathcal{T}_3$ とし,$x_{1-D}^0 = \cdots 010$, $x_1^T = 0110100$ の符号化確率を計算する.

1. まず文脈 '010' に対応した葉を探す.

2. 葉から λ まで文脈に沿ってさかのぼりながら P_e, P_w を計算していく.

$$P_w^{010} = P_e^{010}(1,0) = \frac{1}{2}$$
$$P_e^{10}(1,0) = \frac{1}{2},\ P_w^{10} = \frac{1}{2}P_e^{10} + \frac{1}{2}P_w^{010}P_w^{110} = \frac{1}{2}$$

$$P_e^0(1,0) = \frac{1}{2},\ P_w^0 = \frac{1}{2}P_e^0 + \frac{1}{2}P_w^{10}P_w^{00} = \frac{1}{2}$$
$$P_e^\lambda(1,0) = \frac{1}{2},\ P_w^\lambda = \frac{1}{2}P_e^\lambda + \frac{1}{2}P_w^0 P_w^1 = \frac{1}{2}$$

3. $p_w^\lambda = \frac{1}{2}$ が符号化確率となる.

4. 情報源から発生するシンボルは '0' なのでこのまま次の符号化確率の計算に入る.

同様にして全て計算すると図 1 のようになる.

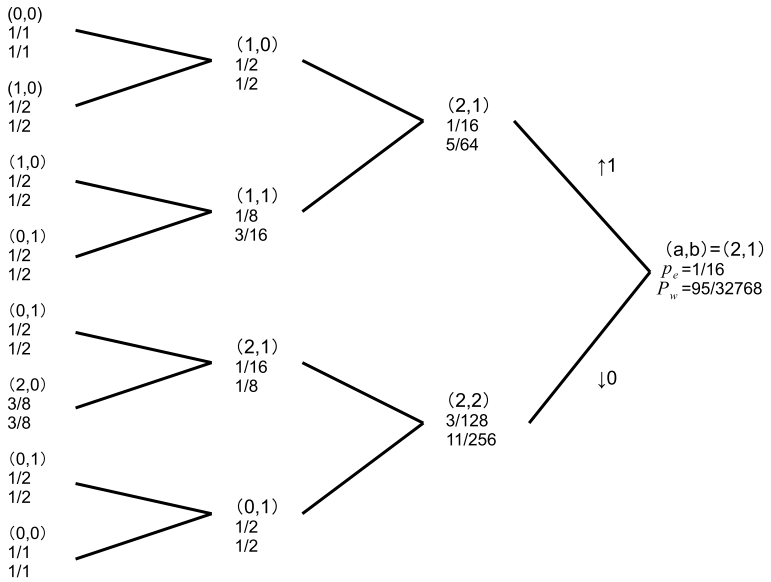

図 1　符号化確率の計算結果

3　圧縮率を算出するにあたっての改良

3.1　符号化確率からの圧縮率の計算

CTW 法は符号化確率を計算し, その符号化確率を用いて算術符号を行う. しかし, 本研究では算術符号を行わずに符号化確率から平均符号語長を計算する. P を符号化確率, を l 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.

$$-\log_2 P + 1 \leq l \leq -\log_2 P + 2 \tag{5}$$

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,

$$\frac{-\log_2 P}{n} + \frac{1}{n} \leq \frac{l}{n} \leq \frac{-\log_2 P}{n} + \frac{2}{n} \tag{6}$$

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,$l = -\log_2 P$ として扱うことができることが分かる.

また, 本論文では圧縮率を 1 シンボルあたりビット数 ([bit/シンボル]) と定義する. 従って圧縮率はこの値が小さいほどよいと言える.

3.2 符号化確率の条件確率

このように符号化確率から圧縮率を計算できるが, このままでは符号化確率は減少していき, 精度が足りなくなってしまう. そこで,Sadakane, Okazaki and Imai が提案した条件付確率を用いて前のシンボルが発生した条件での確率を求める方法 [2] を使う. シンボル x_t が前後関係 $0s$ を持つとする, 前後関係が s のときの $x_t = c$ の重み付け確率は以下のように計算できる.

$$\begin{aligned} P_w^s(x_t = c \mid x_1^{t-1}) &= \frac{P_w^s(x_t = c, x_1^{t-1})}{P_w^s(x_1^{t-1})} \\ &= \frac{\gamma P_e^s(x_1^t) + (1-\gamma) P_w^{0s}(x_1^t) P_w^{1s}(x_1^t)}{\gamma P_e^s(x_1^{t-1}) + (1-\gamma) P_w^{0s}(x_1^{t-1}) P_w^{1s}(x_1^{t-1})} \\ &= \frac{\gamma P_e^s(x_1^{t-1}) P_e^s(x_t = c) + (1-\gamma) P_w^{0s}(x_1^{t-1}) P_w^{0s}(x_t = c \mid x_1^{t-1}) P_w^{1s}(x_1^{t-1})}{\gamma P_e^s(x_1^{t-1}) + (1-\gamma) P_w^{0s}(x_1^{t-1}) P_w^{1s}(x_1^{t-1})} \end{aligned} \tag{7}$$

ここで β を次のように定義する.

$$\beta = \beta_s(x_1^{t-1}) = \frac{P_e^s(x_1^{t-1})}{P_w^{0s}(x_1^{t-1}) P_w^{1s}(x_1^{t-1})} \tag{8}$$

そして, β を用いて $P_w^s(x_t = c \mid x_1^{t-1})$ を表すと,

$$\begin{aligned} P_w^s(x_t = c \mid x_1^{t-1}) &= \frac{\gamma\beta P_e^s(x_t = c) + (1-\gamma) P_w^{0s}}{\gamma\beta + (1-\gamma)} \\ &= \frac{\gamma\beta}{\gamma\beta + (1-\gamma)} P_e^s(x_t = c) + \frac{\gamma\beta}{\gamma\beta + (1-\gamma)} P_w^{0s}(x_t = c \mid x_1^{t-1}) \end{aligned} \tag{9}$$

$P_e^s(x_t = c)$ は前後関係 s までに起こっているシンボルの数によって KT 推定によって計算される. また $\beta_s(x_t) = c$ 以下のように再帰的に計算することができる.

$$\beta_s(x_1^t) = \frac{P_e^s(x_1^t)}{P_w^{0s}(x_1^t) P_w^{1s}(x_1^t)} = \beta_s(x_1^{t-1}) \cdot \frac{P_e^s(x_t = c)}{P_w^{0s}(x_t = c \mid x_1^{t-1})} \tag{10}$$

また β の初期値は 1 である. シンボル x_t は, この条件付確率により算術符号で符号化される.

4　有限メモリにおける文脈木重み付け法

4.1　使用メモリ量を制限しない圧縮

本研究では有限メモリ上での圧縮を目的としているが, 比較のために使用メモリ量を制限しない場合での圧縮率を算出した. なお圧縮を行うファイルはカルガリーコーパス [3] を用いた. 図 2 に文脈木の深さ D に対する圧縮率を示した.

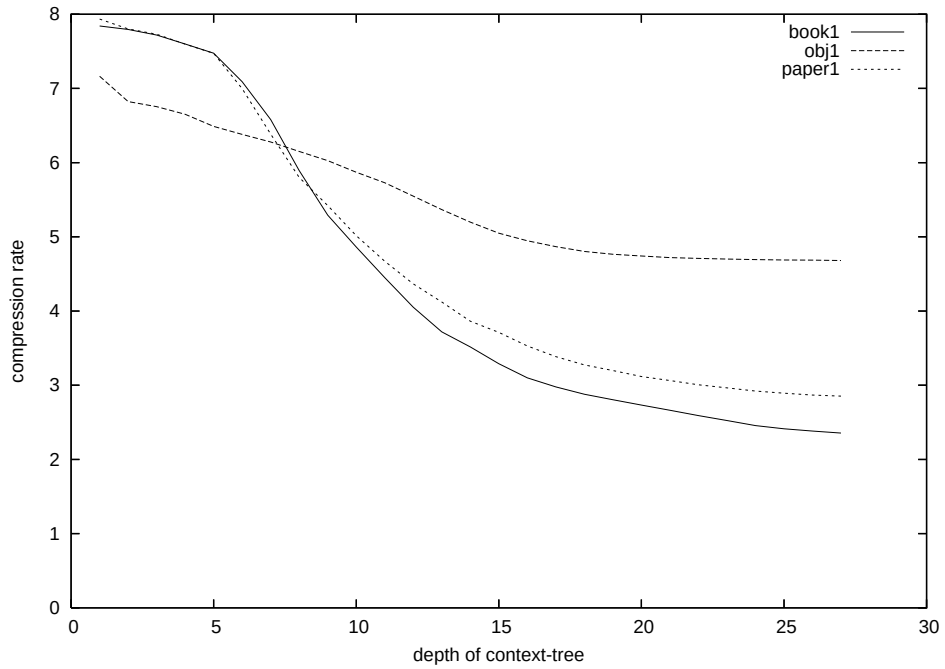

図 2　非有限メモリでの圧縮率

カルガリーコーパスは 18 個のファイルがあるが, 全て同じ傾向が現れたので代表的な物のみ掲載する. 図 2 を見て分かる通り book1,paper1 については $D = 6$ あたりから圧縮率が向上していることが分かる. これはアスキーコードが関係している. これらのファイルはテキストファイルである. そして CTW 法はバイナリの文脈木を用いて行うため, テキストの一文字を 8[bit] に分解して符号化を行う. よって 6[bit] 見るとその文字が何なのかが, 絞り込まれていることが分かる. obj1 についてはバイナリファイルなのでこのような特徴は見られない.

また, 単調減少となっていて D を出来るだけ大きく設定すると, 圧縮率が向上されることが分かる. しかし, 有限メモリでの圧縮には問題点が 2 点ある. 次の節からは問題点とその解決法を提案し, 性能の評価をする.

4.2 カウンタの上限を設定した場合

一つ目の問題点は大きなファイルを圧縮した場合, カウンタ (a, b) が溢れてしまう可能性があることである. この問題を解決するために (a, b) の片方がある値に達したら, (a, b) を $1/2$ にし, 小数点以下を切り捨てる方法を提案する. 式 (2)(3) より P_e が (a, b) の値の割合で算出されることが分かる. よってこの方法で (a, b) を $1/2$ にしても (a, b) の割合が著しく損なわれることはないので, 圧縮率が悪くなることはないと考えられる. ここで表 1 に (a, b) の上限がない場合と, 上限を 20 に設定した場合の比較を示す.

このように (a, b) の上限を 20 にした場合でも, 圧縮率は悪くなったものの 0.099〜0.178 の差しかなく実用的といえる. また, 上限 20 という値は非常に小さい値であり, 上限を大きくすれば差はさらに小さくなる.

4.3 文脈木の深さと圧縮率の関係

CTW 法は文脈木を用い次のシンボルが何かを予測する方法であるが, この方法を用いるには文脈木の各ノードで (a, b),P_e,P_w を記憶しておく必要がある. そこで,(a, b),P_e,P_w を 1 メモリとして文脈木の深さに対する圧縮率とメモリ使用量の関係を図 3 に示す.

図 3 より文脈木の深さ D が大きいほど圧縮率が良くなることがわかる. 従来法では全てのメモリを使用する, しないに関わらず文脈木の深さに応じたメモリを用意しメモリ量を制限している. しかし, そのために用意しなければならないメモリは指数的に増えてしまう. だが実際にはその全てのメモリを使用しているわけではない. つまり, 情報源に出てこない文脈が存在する. さらに使用されないメモリは文脈木の深さに対して急激に増加していく.

今 D を制限することで, 使用可能メモリを制限することは可能だが前記の理由から無駄が存在することが分かる. 次節では無駄を軽減する方法を提案する.

4.4 生成できるノード数に制限を付けた場合の圧縮

前節で示したとおり D を制限することで, 使用可能メモリを制限すると無駄なメモリができてしまうことが分かった. そこで, 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する. さらに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ノードのメモリを消去するようにする. 図 4 に $x_{1-D}^{0} = \cdots 00$, $x_1^T = 11$ の例を示す.

図 4 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. また提案法でも文脈木の深さ D を設定する必要がある. この例では深さ 2 に設定している.

文脈木の深さ D	上限なしの圧縮率	上限 20 の圧縮率	差
3	7.704	7.848	0.143
4	7.509	7.642	0.133
5	7.368	7.496	0.128
6	6.664	6.783	0.120
7	5.861	5.959	0.099
8	5.296	5.406	0.110
9	4.928	5.038	0.110
10	4.484	4.603	0.119
11	4.128	4.256	0.128
12	3.851	3.989	0.138
13	3.665	3.811	0.145
14	3.506	3.656	0.150
15	3.389	3.541	0.152
16	3.243	3.399	0.156
17	3.107	3.266	0.160
18	2.981	3.144	0.163
19	2.910	3.077	0.166
20	2.843	3.012	0.169
21	2.785	2.955	0.170
22	2.729	2.900	0.171
23	2.666	2.838	0.172
24	2.598	2.772	0.174
25	2.548	2.724	0.175
26	2.500	2.677	0.177
27	2.476	2.654	0.178

表 1　カウンタに制限がない場合とある場合の比較 (book1)

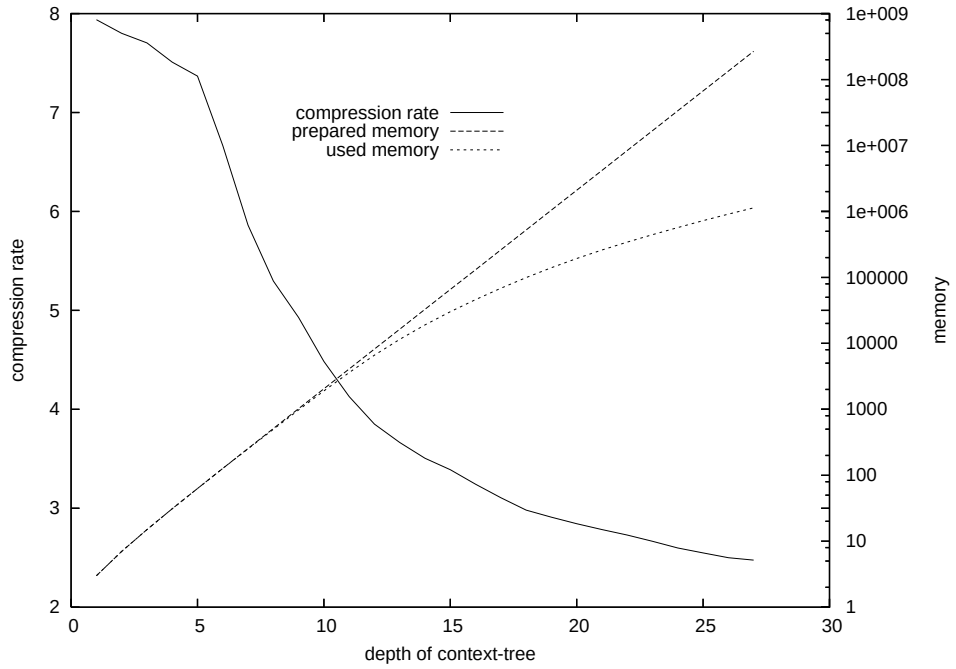

図 3　文脈木の深さに対する圧縮率とメモリ使用量

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．この方法を用いカルガリーコーパスの圧縮を行った結果を図 5 に示す．

図 5 を見ると使用可能メモリを増やすと圧縮率が良くなることが分かる．しかし, 文脈木の深さを制限したときの圧縮率とは異なる結果が得られた．そこで図 6 で文脈木の深さを制限したときの圧縮率と提案法での圧縮率を比べる．

図 6 から図 23 を見ると全てに同様の特徴が見られる．例えば図 7 を見ると, メモリ数 10^5 までは従来の方法の方が圧縮率が良いことが分かる．これは従来法ではメモリ数が少ないときにはほぼ全てのメモリを使用していて無駄が少ないためである．しかし, メモリ数が多い場合では無駄が多くなり, その無駄の分だけ深く文脈木の枝を伸ばすことで, 従来法より提案法の圧縮率が良くなる．また, 提案法ではメモリの使用可能量を増加させると, 急激に従来法における最良の圧縮率に近づいている．これはメモリの使用可能量が増加したため, あまり使われないノードにもメモリが確保されたままになり, 最終的にメモリの削除が起こらなくなっているためである．つまりメモリ 10^6 からは使用していないメモリが存在していることになる．また提案法でも文脈木の深さを指定する必要があり, この実験では $D = 27$ となっている．文脈木の深さ D をさらに大きくし, 使用していないメモリをより深いノードに確保することで, 更に圧縮率を良くすることが出来る．このことから, 従来の方法で無駄になっていたメモリの分だ

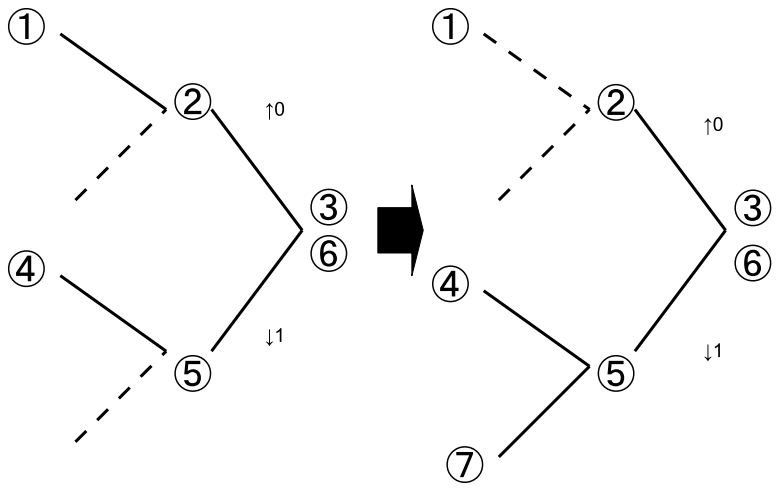

図 4　メモリの削除方法 (メモリ上限 5)

け使用する部分で深くメモリを確保することで圧縮率が改善されることが分かった.

5　まとめ

CTW 法を有限メモリで実現させるための提案, 提案法の性能評価を行った. 第一の提案は,CTW 法で圧縮を行う際に文脈木の各ノードに保持されているカウンタ (a, b) が溢れる可能性がある可能性に着目し, カウンタが上限に達したら (a, b) を $1/2$ にし小数点以下を切り捨てる方法である. この方法を用いて実際にファイルの圧縮を行い比較を行ったところ, 圧縮率はほとんど悪くなることはなく実用的といえた. 第二の提案は, 文脈木をより深く設定することで圧縮率がよくなる性質, 文脈木を深く設定すると使用されないノードが存在することに着目し, 設定したメモリ数の上限に達したら使用したのが一番古いメモリから削除するというものだった. 結果, 無駄になっていたメモリの分だけ使用する部分で深く文脈木を伸ばすことで圧縮率はよくなった.

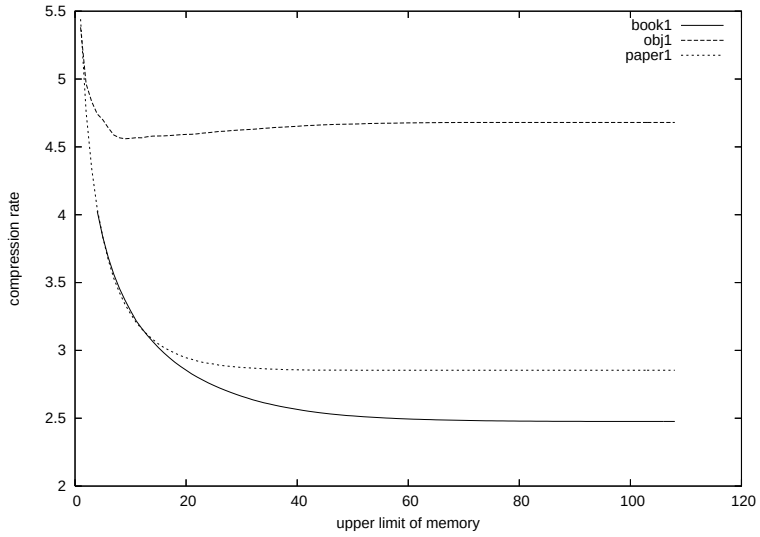

図 5　使用可能メモリに対する圧縮率

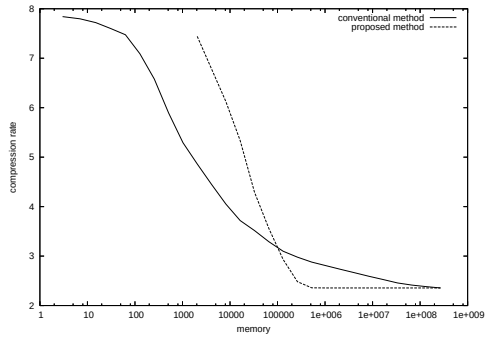

図 6　従来法と提案法の圧縮率比較 (bib)

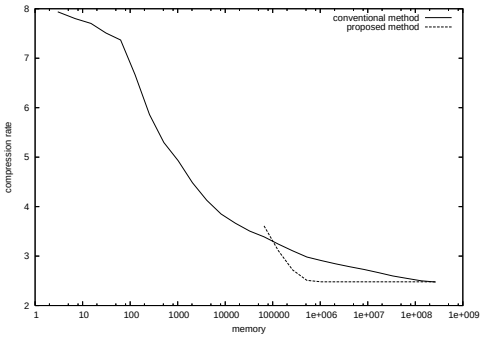

図 7　従来法と提案法の圧縮率比較 (book1)

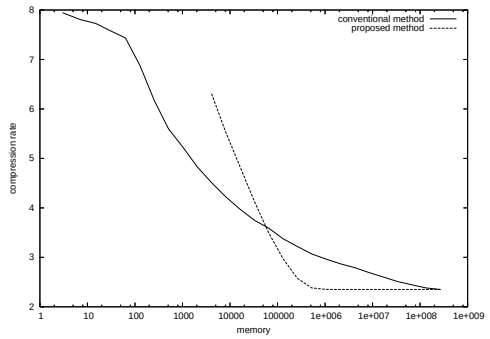

図 8　従来法と提案法の圧縮率比較 (book2)

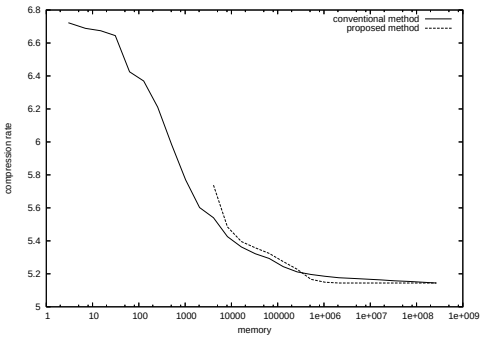

図 9　従来法と提案法の圧縮率比較 (geo)

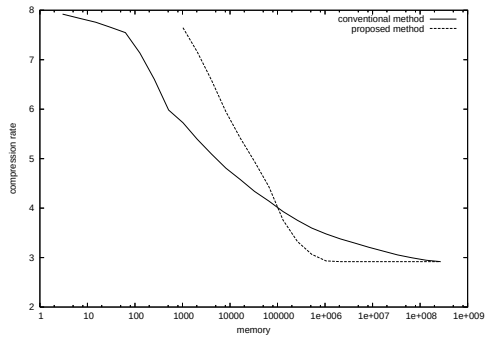

図 10　従来法と提案法の圧縮率比較 (news)

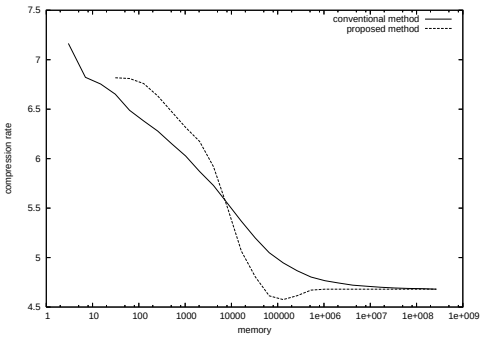

図 11　従来法と提案法の圧縮率比較 (obj1)

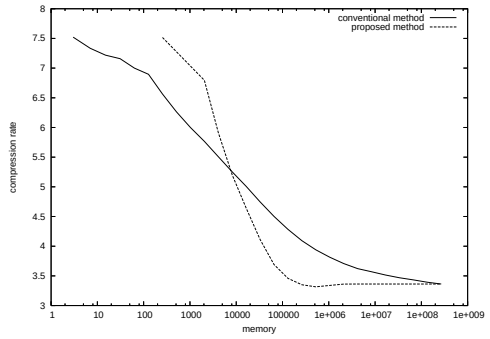

図 12　従来法と提案法の圧縮率比較 (obj2)

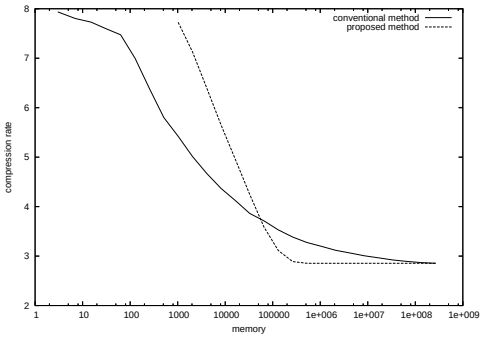

図 13　従来法と提案法の圧縮率比較 (paper1)

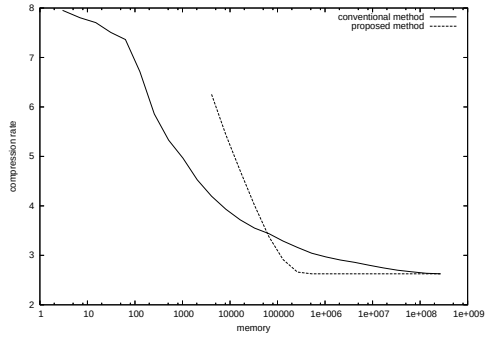

図 14　従来法と提案法の圧縮率比較 (paper2)

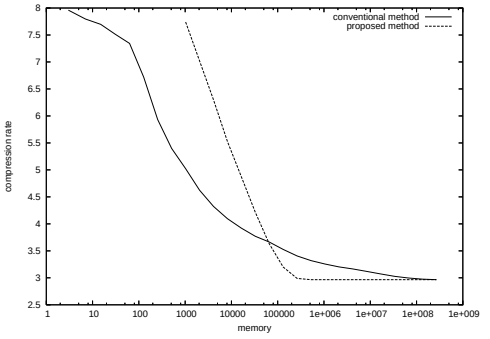

図 15　従来法と提案法の圧縮率比較 (paper3)

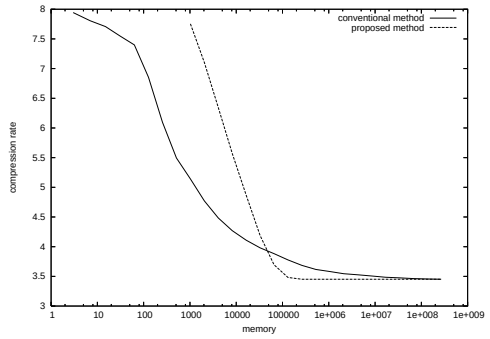

図 16　従来法と提案法の圧縮率比較 (paper4)

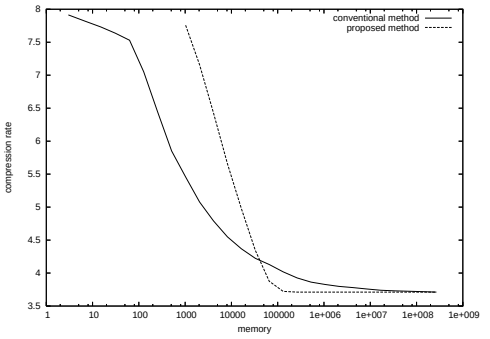

図 17　従来法と提案法の圧縮率比較 (paper5)

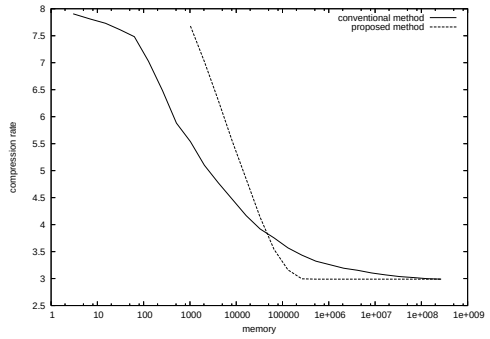

図 18　従来法と提案法の圧縮率比較 (paper6)

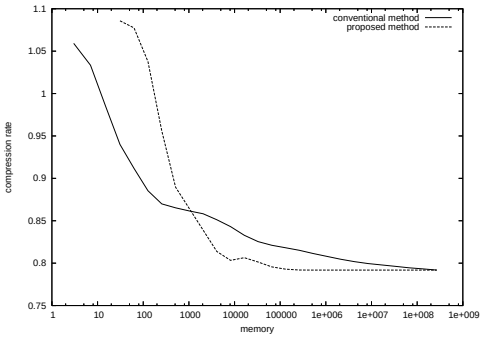

図 19　従来法と提案法の圧縮率比較 (pic)

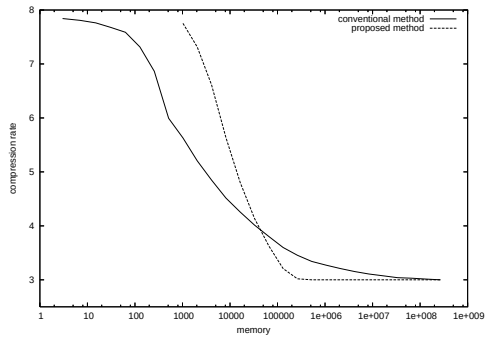

図 20　従来法と提案法の圧縮率比較 (progc)

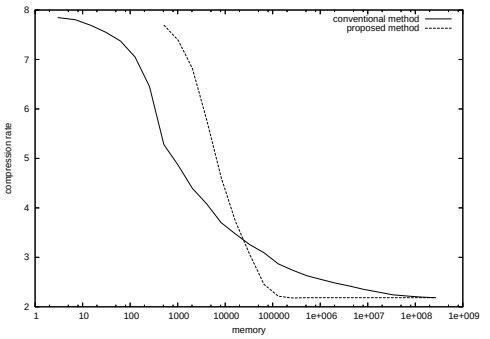

図 21　従来法と提案法の圧縮率比較 (plogl)

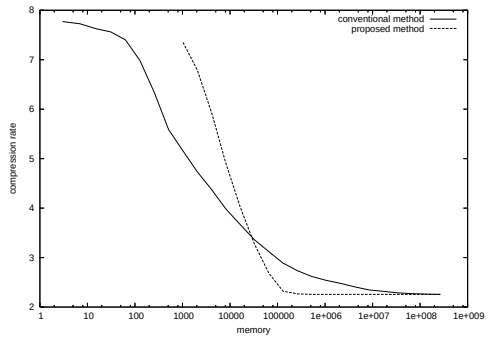

図 22　従来法と提案法の圧縮率比較 (progp)

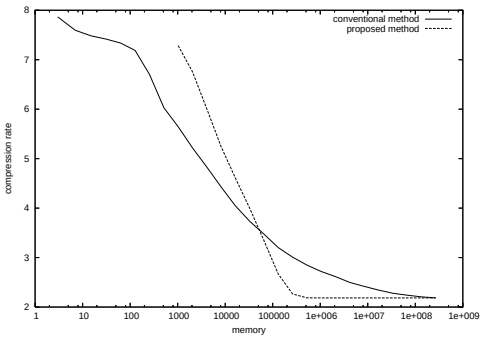

図 23　従来法と提案法の圧縮率比較 (trans)

謝辞

本研究を行うにあたって, 多方面にわたって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] Kunihiko Sadakane, Takumi Okazaki, Hiroshi Imai “ Implementing the Context Tree Weigghting Method for Text Compression ” Data Compression Conference (DCC ’00), pp. 123–132, 2000.

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

[4] 河西朝雄, なぞりがき C 言語学習ドリル, 技術評論社, 2008.

付録 A　生成できるノードに制限を付けた場合での圧縮率算出プログラムのソースコード

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

#define DEPTH 27

int max_nodes;

typedef struct node{
    long a;
    long b;
    double pe;
    double pw;
    double tpe;
    double tpw;
    double beta;
    int newer;
    int older;
} node;
node *branches[(1UL << (DEPTH + 1)) - 1];
int newest;
int oldest;
char history[DEPTH + 1];
long number_of_nodes;


int findleaf()
{
    int n;
    int i;
    n=0;
    for (i = 0; i < DEPTH; i++){
        if (history[i] == '0'){
            n = 2*n+1;
        } else {
            n = 2*n+2;
        }
    }
    return(n);
}

node *alloc_node(int n)
{
    node *nd;

    if (number_of_nodes >= max_nodes){
        int deleted = oldest;
        if (branches[oldest]->newer == -1){
            newest = branches[oldest]->older;
        } else {
            branches[branches[oldest]->newer]->older = branches[oldest]->older;
        }
        oldest = branches[oldest]->newer;
        free(branches[deleted]);
        branches[deleted] = NULL;
        number_of_nodes--;
    }
    nd = (node *)malloc(sizeof(node));
    if (nd == NULL){
        printf("malloc() error");
```

```
        exit(1);
    }
    nd->a = 0;
    nd->b = 0;
    nd->pe = 1.0;
    nd->pw = 0;
    nd->tpe = 0.5;
    nd->tpw = 0.5;
    nd->beta = 1.0;
    nd->newer = -1;
    if (newest == -1){
        newest = n;
        nd->older = -1;
        oldest = n;
    }else{
        branches[newest]->newer = n;
        nd->older = newest;
        newest = n;
    }
    number_of_nodes++;
    return(nd);
}

void listupdate(int 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(char update)
{
    int n;
    int d;
    double prev_tpw = 0.0;

    for (d = DEPTH, n = findleaf(); d >= 0; d--, n = (n - 1) / 2){
        if (branches[n] == NULL){
            branches[n] = alloc_node(n);
        } else {
            branches[n]->tpe = (((update == '0')? branches[n]->a: branches[n]->b) + 0.5)
                               /(branches[n]->a + branches[n]->b + 1.0);
            if (update == '0'){
                listupdate(n);
            }
        }
        if (d == DEPTH){
            branches[n]->tpw = branches[n]->tpe;
        } else {
            branches[n]->tpw = branches[n]->beta / (branches[n]->beta + 1.0)
                               * branches[n]->tpe + 1.0 / (branches[n]->beta + 1.0) * prev_tpw;
        }
        prev_tpw = branches[n]->tpw;
    }
    return;
}
```

```
void actualupdate (char t)
{
    int d;
    int n;
    for (d = DEPTH, n = findleaf(); d >= 0; d--, n = (n - 1) / 2){
        branches[n]->pe = branches[n]->tpe;
        branches[n]->pw = branches[n]->tpw;
        if (t == '0'){
            branches[n]->a++;
        } else {
            branches[n]->b++;
        }

        if (d < DEPTH){
            branches[n]->beta = branches[n]->beta * branches[n]->pe
                                / branches[((history[d] == '0')? 2 * n + 1: 2 * n + 2)]->pw;
        }
    }
    return;
}

void historyupdate(char t)
{
    int i;
    for (i = DEPTH - 2; i >= 0; i--){
        history[i+1] = history[i];
    }
    history[0] = t;
    return;
}

void inithistory(void)
{
    int i;

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


main(int argc, char **argv)
{

    FILE *fp;
    int c;
    int mask;
    char t;
    double length = 0.0;
    int j = 0;


    number_of_nodes = 0;
    newest = -1;
    oldest = -1;

    if (argc != 2) goto ERROR;
    if (argv[1] == NULL) goto ERROR;
    if ((max_nodes = atof(argv[1])) == 0.0) goto ERROR;

    inithistory();
    if ((fp = fopen("book1.dat", "rb")) == NULL){
        printf("fileopn error\n");
    exit(1);
    }
```

```
    while ((c = getc(fp)) != EOF){
        for (mask = 1 << 7; mask != 0; mask >>= 1){
            t = (c & mask)? '1': '0';
            dummyupdate('0');

            if (t == '1'){
                dummyupdate('1');

            }
            actualupdate(t);
            length = length - log10(branches[0]->pw)/log10(2.0);
            j = j + 1;

            historyupdate(t);
        }
    }
    length = 8.0 * length / j;

    ERROR:exit(1);

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