自動的なパターン抽出によるデータ圧縮法の提案†

河村 知行††

データの圧縮法には，これまでにも種々なものがあった．それらは，おもにデータ通信の技術の一部として，高速でかつリアルタイムな処理を目的としていた．本論文では，高速性やリアルタイム性は考えずに，データの圧縮率に的をしぼったデータ圧縮法であるパターン抽出法 (Pattern Extraction Method : PEM) を提案し，その評価を行っている．結論として，これまでの圧縮法より，圧縮率において，約40パーセント良い結果が得られた．

1. はじめに

データ圧縮については，これまでにもさまざまな方法が提案されている．そのうちには，①各文字の頻度の片寄りにより，出現頻度の高い文字には短いコードを割り付ける方法（ハフマン符号）[1]，②文字コード表の未使用の部分に，出現頻度の高い文字の2文字の組合せを，1文字として割り付ける方法[2]，③同一文字の連続した連なりを，連続する文字と，その長さにより表現する方法（ランレングス符号）[1]，④文字列中に存在する同一の文字列を，逐次蓄積していく方法[3]などがあった．①②は一度，各文字の出現頻度を知れば，リアルタイムな処理（圧縮結果を逐次出力すること）が可能であるが，圧縮の効果はあまり上がらない．また，各文字の出現頻度が異なるデータについては，その出現頻度を別な情報としてもつ必要がある．③も，リアルタイムな処理が可能である．この方法は，同一文字の連なりが頻発するデータには有効であるが，一般の文字データに，そのことは期待できない．

④は，増分記号列逐次蓄積法と呼ばれる方法で，データの処理は，前述の①～③がリアルタイムであったのに対し，バッチ処理となる．ただし，詳しくは第5節で述べるが，圧縮結果は，前から逐次確定していくので，処理結果の出力は，逐次出力可能である．

本論文で報告する新しいデータ圧縮法は，上述の中では，④に近いもので，完全なバッチ処理によりデータ圧縮が行われる．処理時間を多く必要とする代わりに，高い圧縮率を得ることができる．本圧縮法を，パターン抽出法（PEM : Pattern Extraction Method）と呼ぶ．

† New Data Compression Method by Automatic Pattern Extraction by TOMOYUKI KAWAMURA (Department of Information Electronics, Tokuyama Technical College).
†† 徳山工業高等専門学校情報電子工学科

2. パターン抽出法（PEM）のアルゴリズム

パターン抽出法のアルゴリズムを説明する．

この節では，直観的な理解を容易にするために，JIS7コードの文字の集まりをデータとして取り扱う．JIS7コードは7ビットコードであるが，8ビットの領域（バイト）に格納されるものとする．最上位のビットは，PEM が作り出す新しいコードのために使われる．これらのコードを，以下，〈128〉〈129〉…〈255〉と記し，圧縮コードと呼ぶ．〈128〉は，文字列の区切りとしての特別な意味をもち，$ で記すこともある．

PEM は，

AAAAAAAABBCAAAAAAAABBD

なる文字列を最終的に，

〈129〉C〈129〉D$〈130〉〈130〉BB$AAAA$

のようなコードの列に置きかえる．その変化の過程を**図1**に示す．

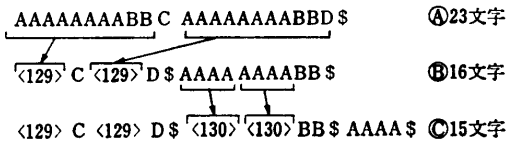


図1 PEM による圧縮の過程
Fig. 1 Process of compression by PEM.

図1の矢印が，新しくコード化された場所を示している．この例では最終圧縮率は 15/23=0.652 になっている．各段階における文字列（＝コード列）の最後の「$」は省くことも可能であるが，一貫性のためにつねに含めて議論を行う．

さて，図1Ⓐから Ⓑへの変換において，圧縮コード〈129〉で示されるパターン AAAAAAAABB を抽出する方法を説明する．ここが PEM の中心的部分である．**図2**にパターンを抽出する過程を示す．

図2は，文字列と数の集合を一つの節点とするツ

リー構造となっている．文字列は，そこまでに抽出されたパターンを表す．数は，その節点のパターンの最後の文字の位置を示す．同じパターンが2回以上現れる場合は，そのすべての位置が示されている．このツリーをPEMツリーと呼ぶ．また，各節点のパターンの長さを節点の長さと呼ぶ．図2の中で実線で囲まれた三つの節点に注目してみる．上の節点は，AAAAという長さ4のパターンが，4 5 6 7 8 15 16 17 18 19の位置にあることを示す．このとき，各位置から右の1文字を取り込んだパターンを発生させると，実線内の下の二つの節点が作り出される．すなわち，AAAAA という長さ5のパターン（位置は 5 6 7 8 16 17 18 19）と，AAAAB という長さ5のパターン（位置は[9 20）である．一般に下側の節点の個数は，上の節点が新しく取り込んだ文字の種類数である．この例の場合，文字の種類はAとBだけであったので，下側の節点の数は2となっている．この下側の節点を作り出す操作をα展開と呼ぶ．α展開の上の節点のパターンの位置の数と，下側の各節点のパターンの位置の数の合計とは，特殊な場合を除き一致する．特殊な場合とは，取り込んだ文字が区切コード〈128〉"$"の場合である．〈128〉を含んだパターンは生成しないようにする．さらに，各節点のうち，パターンの数が1個になってしまったものは，α展開を行わないようにする．

与えられた文字列Sから図2のようなPEMツリー全体を作り出すことをβ展開と呼ぶ．また，文字列Sに対して，そのPEMツリーの節点の集合を$\beta(S)$で表す．

$\beta(S)$のなかから，何らかの基準により，一つの節点を選び出す関数をVとする．また，文字列Sのなかの，パターンPをすべて圧縮コードCで置きかえ，その後にパターンPと区切コード〈128〉を付け加えた文字列を値とする関数を COMP(S, P, C) とする．

以上の準備により，PEMは次のように記述できる．「与えられた文字列Sから，PEMツリー$\beta(S)$を作り，そのなかから，ある基準で，一つのパターンPを選ぶ．SのなかのPを新しいコードCで置きかえて，それを新しいSとする．以上の処理を，Pが選べなくなるまで繰り返す．」

これを，アルゴル風の言語で記述すると，**図3**のようになる．

図1の例では，文字列Ⓐのなかから，AAAAAAAABB が選ばれ，〈129〉により置きかえられ，文字列Ⓑとなる．文字列Ⓑから，AAAA が選ばれ，〈130〉により置きかえられ，文字列Ⓒとなる．

以上が圧縮の過程である．

一方，圧縮された文字列から元へ戻すには，まず，区切コード〈128〉を探すことにより，各パターンの圧縮コードが決まる．次に，圧縮された文字列の先頭から，最初の区切コードまでに注目する．このコード列中の圧縮コードを復元することにより，元の文字列が得られる．圧縮コードが示すコード列のなかに，他の圧縮コードが入っていることもあるので，復元プログラムは再帰的なプログラムとなる．

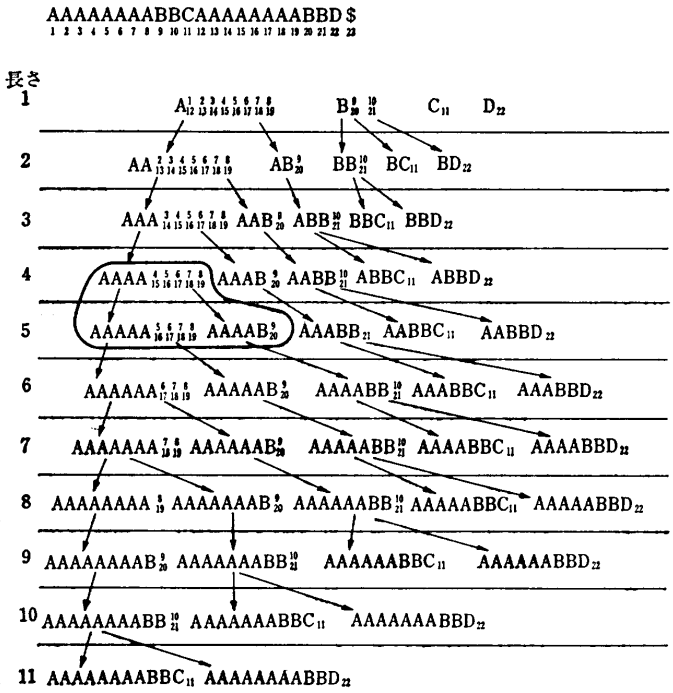


図 2 PEM ツリーの例
Fig. 2 Example of PEM tree.

```
S のセット ($ を含む)
C: =128
loop
  P: =V(β(S))
  if P=nil then exit
  C: =C+1
  S: =COMP (S, P, C)
endloop
S の出力
```

図 3 PEM のアルゴリズムの概略
Fig. 3 Outline of PEM algorithm.

一般に，PEM は，圧縮では，バッチ処理でかつ大量の処理時間と作業領域を必要とするが，復元では，小量の作業領域で，高速な復元が可能である．

3. PEM のインプリメント

PEM プログラムの最大のポイントは，PEM ツリーの作成にある．図2にある PEM ツリーの各節点は，パターンの長さも位置の個数も可変であり，単純なインプリメントでは莫大な記憶領域を必要としてしまう．そこで，次の2点の工夫が行われている．

① もとの文字列（図2の例で長さ 23）を1次元配列に置いて，各節点は，パターンの長さと位置の集合により表現している（パターンそのものは不必要）．圧縮コードが256以上になることがあるので，配列の要素の大きさは10ビット程度必要である．実際には2バイトを割り付けてある．

② PEM ツリーの作成（β 展開）と，PEM ツリーから節点を選ぶ選択関数(V)とを一体の手続として行い，PEM ツリーの枝刈を行っている．たとえば，一度しか現れないパターンは，コード化する利益がないので，いかなる関数Vでも，PEM ツリーの葉を選択することはない．すなわち，すべての関数 V で，PEM ツリーの葉を記憶領域から削除することが可能である．

上の二つの工夫に加えて，今回作成したプログラムでは，値はすべてリストにより管理して，不必要になった記憶領域を，可能な限り早期にかつ高速に回収するようにした．また，PEM ツリーの作成は横型探索により行った．すなわち，長さnの節点をすべて発生させてから，長さ $n+1$ の節点を作っていく．これにより，節点の長さの格納領域も，大域変数一つで済ませている．さらに，横型探索では，図2の下向きのポインタの代りに，同じ長さの節点を横につなぐポインタが必要であるが，このポインタも回収が可能になり次第，回収している．

以上に示した PEM ツリーを作成していく過程で，ある基準で各節点を評価し，その時点までにおける最も評価のよい節点を覚えておくようにしている．PEM ツリーの作成が終わった時点での，最も評価のよい節点が選択関数Vの値となる．

何回かの繰返しの後，選択関数の値が nil になり，最終的に圧縮が完了したとき，各コード（2バイト）をファイル等に格納するのであるが，小さなコードは1バイトで，大きなコードは2バイトで格納している．そのために B2POINT と呼ぶ値を用いている．現在のインプリメントでは，B2POINT の値は 252 である．すなわち，252 未満のコードはそのまま1バイトで格納し，252以上のコードxは，[$(x-252)$ DIV 256+252] [$(x-252)$ MOD 256] の2バイトを格納している．これにより，128 から $1275=256\times(256-252)+252-1$ までの圧縮コードを作り出すことができ，かつ，頻発する小さなコードには，1バイトの格納領域ですませることができる．

4. 選択関数

今回の実験は，次の四つの選択関数について行った．

① 圧縮率優先選択関数
② 個数優先選択関数
③ 長さ優先選択関数
④ 減少文字数優先選択関数

4.1 圧縮率優先選択関数

圧縮率優先選択関数とは，各節点が含むすべてのパターンを圧縮コードで置き換えたとき，最も小さな圧縮率をとる節点を，選択関数の値とする．圧縮率 (R) は，次の式で計算される．

$$R=\frac{PN*CL+PL+DL}{PN*PL}$$

PN：パターンの個数，CL：圧縮コードの長さ，
PL：パターンの長さ，DL：区切りコードの長さ

すべての節点の圧縮率が1以上のとき，選択関数の値は nil である．ある節点のなかのパターンが文字列中で，互いに重複している場合，PN は，重複しないパターンの数とする．新しい圧縮コードの長さや，区切コードの長さは，ふつう1（1バイト）である．ただし，圧縮コードは，B2POINT 以上になると長さ2となる．

α 展開において，下側にある節点のパターンの数は，上にある節点のパターンの数以下である性質（パターン数の単調減少）を用いて，式Rの中のパターンの長さを無限大とすることにより，その節点から派生する節点の圧縮率の理論的最小値を求めることができる．この値が，その時点までにおける最も小さな圧縮率以上であれば，その節点はこれ以上生長させても無意味なので，枝刈をすることができる．実際のプログラムとしては，パターンの長さは無限大とはせずに，長さ200として，枝刈が起きやすくしている．

4.2 個数優先選択関数

個数優先選択関数は，各節点の含むパターンの数が

最も多い節点を選択関数の値とする．ただし，この条件だけでは，つねに長さ1の節点が選択されてしまう．そこで，「その節点の圧縮率が1未満である」という条件を付加している．条件を満たす節点がないとき選択関数の値は nil となる．

この方法は多くの場合，長さ2のパターンが選択されて，効率が悪いようにも見えるが，「α 展開によるパターン数の単調減少」の性質により，多くの場合は，長さ3以上の節点はすべて枝刈ができるという長所をもつ．

4.3 長さ優先選択関数

長さ優先選択関数は，各節点のパターンの長さが最も長い節点を選択関数の値とする．これにも，圧縮率が1未満の節点のなかで選択するという条件を付加している．条件を満たす節点がないとき，選択関数の値は nil である．

4.4 減少文字数優先選択関数

減少文字数優先選択関数は，各節点が含むすべてのパターンを圧縮コードで置換えたとき，減少する文字数が最大になる節点を選択関数の値とする．減少文字数Dは，次の式で計算される．

$$D=(PN*PL)-(PN*CL+PL+DL)$$

すべての節点の値Dが0以下のとき，選択関数の値を nil とする．

枝刈のために，パターンの長さを無限大とするのは，式Dから無意味なので，代りに，予想されるパターンの最大の長さを200として，式Dを計算している．この値が，その時点までの最大減少文字数以下であるならば，その節点を枝刈している．

この方法は，圧縮率優先選択関数と似ているが，式Dを計算するのに割算を用いなくてよいので，PEM のハードウェア化のときに有望である．

4.5 その他の変形

PEM では，PEM ツリーを作るのに多くの時間を費やしている．一つの PEM ツリーで，節点（圧縮パターン）を一つだけ選択するのは非能率である．そこで，一つの PEM ツリーで複数個の節点を選択する方法が考えられる．そのためには，たとえば「それまでの最良の節点10個をソートして記憶しておき，新しい節点が10番目の最良節点よりよい節点であれば，古い10番目の節点を捨てて，新しい10個の節点をソートしていく」などの方法がある．しかし，この方法では，一般に PEM による最適解が得られないので，今回の実験では行っていない．

5. 評　　価

四つの選択関数によって，図1に挙げた文字列が圧縮される過程を**図4**に示す．

各過程の最後に書かれた「数→数」は，最初の文字数と圧縮後の文字数を示している．図1に上げた例は，実は長さ優先の圧縮であった訳である．

次に実用的テキストに対して，PEM を適用してみる．テキストは，PASCAL 言語によるプログラム (PRG) と，一般の英文 (DOC)，おのおの3通り，計6個のデータについて実測を行った．その結果を**表1**に示す．比較のため，増分記号列逐次蓄積法* と，ハフマン符号による圧縮率も添えてある．

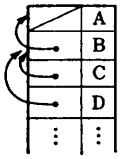
最終圧縮率は，最終コード列の長さ／最初の文字列の長さ，で計算している．最終コード列の長さは，B2POINTを用いて，大きな圧縮コードは2バイトとして計算している．ハフマン符号の最終圧縮率には，符号と元符号の対応表の大きさは含まれていない．最後の圧縮コードとは，図3のCの最後の値である．最大リストセル数は，できるだけ迅速なセルの回収を行っても，なお必要であったセル数の，瞬間的最大値である．この値は，最初の文字列の長さの約1.05倍であった．計算に必要な主記憶領域の量は，

〔1〕 圧縮率優先（減少文字数優先はこれと同じ）
AAAAAAAABBCAAAAAAAABBD$
⟨129⟩⟨129⟩BBC⟨129⟩⟨129⟩BBD$AAAA$
⟨130⟩C⟨130⟩D$AAAA$⟨129⟩⟨129⟩BB$
23→15

〔2〕 個数優先
AAAAAAAABBCAAAAAAAABBD$
⟨129⟩⟨129⟩⟨129⟩⟨129⟩BBC⟨129⟩⟨129⟩⟨129⟩⟨129⟩BBDAA
⟨130⟩⟨130⟩BBC⟨130⟩⟨130⟩BBDAA⟨129⟩⟨129⟩$
⟨131⟩C⟨131⟩DAA⟨129⟩⟨129⟩$⟨130⟩⟨130⟩BB$
23→16

〔3〕 長さ優先
AAAAAAAABBCAAAAAAAABBD$
⟨129⟩C⟨129⟩D$AAAAAAAABB$
⟨129⟩C⟨129⟩D$⟨130⟩⟨130⟩BB$AAAA$
23→15

図4 四つの選択関数による圧縮の過程
Fig. 4 Process of compression by 4 selection functions.



* 増分記号列逐次蓄積法では，AABABCABD のような文字列から，左のような表を作り，完成した表が圧縮の結果である．ポインタ部は一般に2バイト必要であるが，ある程度は1バイトとして記憶することができる．文字部分はつねに1バイトとして記憶する．

表 1 データ圧縮法の比較
Table 1 Comparison of data compression methods.

データ名	PRG 1	PRG 2	PRG 3	DOC 1	DOC 2	DOC 3
文字数	1510 バイト	4712 バイト	15130 バイト	1560 バイト	4339 バイト	14854 バイト
圧縮率優先 PEM						
最終圧縮率	0.430	0.296	0.306	0.623	0.516	0.499
最後の圧縮コード	178	251	378	203	251	457
最大リストセル数	1609	4859	15360	1666	4461	15042
個数優先 PEM						
最終圧縮率	0.505	0.364	0.334	0.664	0.564	0.549
最後の圧縮コード	244	319	558	229	296	726
最大リストセル数	1609	4859	15360	1666	4461	15042
長さ優先 PEM						
最終圧縮率	0.456	0.399	0.399	0.675	0.667	0.598
最後の圧縮コード	209	313	740	251	328	820
最大リストセル数	1609	4978	15572	1666	4541	15556
減少文字数優先 PEM						
最終圧縮率	0.429	0.294	0.286	0.632	0.519	0.491
最後の圧縮コード	178	251	354	200	251	424
最大リストセル数	1609	5095	15719	1666	4461	15054
増分記号列逐次蓄積法						
最終圧縮率	0.765	0.633	0.537	0.829	0.779	0.742
ハフマン符号						
最終圧縮率	0.612	0.607	0.574	0.563	0.569	0.628

最初の文字列の長さ×2（テキストを収める配列の要素の大きさ：単位バイト）＋（最大リストセル数×4（リストセルの大きさ：単位バイト）

であり，最初の文字列の長さの約6.2倍の主記憶領域が必要である．

結論として PEM は，他の圧縮法より優れた圧縮効果を示す．そのなかでもとくに，圧縮率優先と減少文字数優先による PEM がよい効果を示すことがわかる．

6. PEM の拡張

前章まで述べた PEM は，JIS 7 コードについての圧縮法であったが，PEM はその原理から，最初の文字列の要素の大きさが，任意のビット長であっても適用可能である．

たとえば，EBCDIC のような 8 ビットコードの場合には，区切りコードを 256，B 2 POINT を 508 などとすればよい．そして，補助記憶媒体に格納する際に，9 ビットを 1 記憶単位と考えて，8 記憶単位 72 ビットを 9 バイトに格納するようにすればよい．

さらに，区切りコードと B 2 POINT を用いたインプリメントでは，これらの機能のために，特定のビットを使用するわけではないので，要素の大きさが n ビット（n：整数）である必要はない．すなわち，要素の値が $0 \sim 2^n-1$ でなく，$0 \sim m$（m：整数）であるような配列であっても，効率よく圧縮できるようになっている．

また，PEM は，要素の 1 次元的な並びを圧縮するために開発された手法であるが，2 次元的な並び，すなわち，画像データなどにも適用の可能性がある．要素の 1 次元的な並びと，2 次元的な並びに対するパターンの生長のさせ方を**図 5** に示す．

図 5 のように，うずまき状にパターンを生長させれ

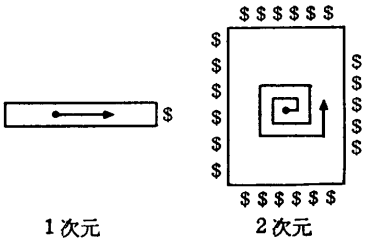


図 5 パターンの生長のさせ方
Fig. 5 Method of growing of pattern.

ば，PEM ツリーを作ることは可能なのであるが，圧縮コードによる詰込み（図3の COMP 操作）が2次元的並びでは不可能である．PEM の2次元的並びへの拡張は，この点を解決しなければならない．

7. まとめ

PEM は，選択関数が異なれば，最終圧縮文字列も異なったものになる．そして，選択関数については，上述のように，いくつかの関数が考えられるが，どのような関数が最適関数なのか明らかではない．これは今後の研究課題であろう．ただし，どのような選択関数で作られた圧縮文字列も，第2章の最後に述べた復元手続きで復元することが可能である．

また，PEM は，かなり多くの主記憶領域と計算時間を必要とする．記憶領域に関しては，今後の LSI 集積度の向上に期待するとしても計算時間に関しては，専用ハードウェアの設計が有効であろう．これが実現すれば，一般の大量データ通信，大量データ記憶への適用も可能になると考えられる．

参考文献

1) 宮川 洋，原島 博，今井秀樹：情報と符号の理論，p. 266，岩波書店，東京 (1982).
2) 坪田信孝，奥田久徳：数字向きバイト単位コードを用いた連接数字圧縮法，情報処理学会論文誌，Vol. 24, No. 6, pp. 727-734 (1983).
3) 日本ユニバック：Compress of 1100, p. 32，日本ユニバック，東京 (1983).

(昭和59年3月9日受付)
(昭和59年6月19日採録)