論文

LZW 法に移動窓を組み合わせたデータ圧縮方式について

正員 荻原 剛志[†]　非会員 飯田 元[††]
正員 井上 克郎[††]　正員 鳥居 宏次[†††]

Compression Method Using LZW Coding and a Sliding Window

Takeshi OGIHARA[†], *Member*, Hajimu IIDA[††], *Nonmember*,
Katsuro INOUE[††] *and* Koji TORII[†††], *Members*

あらまし　データ圧縮法として広く利用されているLZW法に数kByte程度の移動窓（sliding window）を付加し，移動窓中で一致した文字列の長さをLZW符号の代わりに符号化することによって圧縮効率を向上させる方法を提案する．この方式では，LZW法のフレーズを構成する際，移動窓内でのそのフレーズの位置を記録しておく．この情報を利用すると，新しく構成されるLZW法のフレーズが既に移動窓内にあることがわかる場合がある．新たな入力文字列がLZW法のフレーズより長く移動窓内の文字列と一致する場合，LZW符号の代わりに一致する文字列の長さを符号化する．この方式は，大部分の圧縮対象についてLZW法の結果を改善するほか，他のLZW法の改良アルゴリズムと同等以上の性能を示す．処理時間もLZW法に比べて25%程度の増加にとどまり，実用性の高い圧縮法である．

キーワード　データ圧縮，無ひずみ圧縮，LZW法，移動窓

1. まえがき

データ圧縮アルゴリズムとして，さまざまな方式が提案・研究されているが，何を圧縮対象とするのか，あるいは処理速度，圧縮率，必要なメモリ量のどれを重視するのかによって，さまざまな選択肢がある[(1),(2)]．その中でもZivとLempelの提案したインクリメンタル分解[(3)]の実現の一つであるLZW(Lempel-Ziv-Welch)法[(4)]は，入力の性質に依存しない（ユニバーサル性），高速，高圧縮の実用的な無ひずみ圧縮アルゴリズムとして著名である．LZW法はすべての点について他の方法より優れているというわけではないが，処理時間と圧縮能率のバランスという点で最も強力かつ実用的なアルゴリズムの一つであると言うことができよう．

しかしLZW法は，出現回数が多くない文字列は，それが長ければ長いほど，直ちには効率の良い圧縮ができないという欠点をもつ．これは文字列の置換による符号化方式[(5)]（以下ではLZ77法と呼ぶ）と比較した場合に顕著である．この欠点を補うため，LZW法の辞書の成長速度を加速する改良方式[(6)]がいくつか提案されているが，辞書が飽和する時期も早くなるため，入力長が大きい場合には効果が薄い．

一方，LZ77法に基づく圧縮法は，置換文字列の符号化などを工夫することによって高い圧縮能力を実現できるが，移動窓（sliding window）から最も長く一致する文字列を発見する部分が処理効率を悪化させている．

本論文では，LZW法に数kByte程度の移動窓を付加し，移動窓の中で一致した文字列の長さをLZW符号の代わりに符号化する新しい符号化方式とそのアルゴリズムを提案する．この方法によって上記の欠点を解消し，圧縮能力を向上できることを示す．この方式はLZW法とLZ77法の特徴を併せもち，幅広い種類の入力に対して高い圧縮能力を発揮する．LZW法と文字列置換の融合方式であるため，ユニバーサル性は保持している．また，このアルゴリズムはLZW法と同様，入力長に対する線形時間で処理を行い，移動窓のサイズが実行速度に影響しないという特徴も備えてい

† 大阪大学情報処理教育センター，豊中市
Education Center for Information Processing, Osaka University, Toyonaka-shi, 560 Japan
†† 大阪大学基礎工学部情報工学科，豊中市
Faculty of Engineering Science, Osaka University, Toyonaka-shi, 560 Japan
††† 奈良先端科学技術大学院大学情報科学研究科，生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630 Japan

る．処理時間は LZW 法に比べて 25%程度の増加にとどまる．

本方式は LZW 法の高速な処理速度という特長を残しつつ，圧縮効率を向上させることが可能である．従って，現在 LZW 法が用いられている多くの局面に対して実用的に適用が可能であると考えられる．また，移動窓の付加という手法はいくつかの LZW 法の改良方式に対しても同様に適用が可能であり，更に圧縮率を改善することが見込める．

以下では，新たな符号化方式とアルゴリズムについて述べ，圧縮能力に関する実験の結果から，本方式が LZW 法の改良として実用的な方式であることを示す．

2. LZW 法と LZ77 法

2.1 LZW 法の検討

LZW 法による圧縮アルゴリズムのあらましを C 言語の構文を用いて記述したものを図 1 に示す．ここでは，通常の文字はそれ自体の文字コードを符号とし，新たに登録される文字列（フレーズ）には文字コード以上の整数値を逐一割り当てて符号とする．これは UNIX のツールである compress† などの採用している方法である．

関数 InputChar() は 1 文字の読込み，OutputBits (*str*) は符号 *str* の書出しを表す．また，Search(*str*, *ch*) は符号 *str* と文字 *ch* から構成されるフレーズが既に辞書に登録されていればその符号を返し，まだ登録されていない場合には UNUSED（符号として用いない値）を返す．Update(*str*, *ch*) は，符号 *str* と文字 *ch* から構成される新しいフレーズを辞書に登録する．

LZW 法では，長さ n の文字列がフレーズとして登録されるまでに（同じ部分文字列をもつ別の文字列の存在を仮定しなければ）(n-1) 回の出現が必要である．このことは，入力列のサイズが小さい場合には十分な圧縮能力が得られないことを意味する．

```
Compress()
{
    Initialize();
    string = InputChar();
    character = InputChar();
    while ( character != END_OF_STREAM ) {
        idx = Search( string, character );
        if ( idx != UNUSED )
            string = Code( idx );
        else {
            OutputBits( string );
            Update( string, character );
            string = character;
        }
        character = InputChar();
    }
    OutputBits( string );
}
```

図 1 LZW 法のアルゴリズム
Fig. 1 LZW algorithm.

一般にインクリメンタル分解の圧縮方式では，辞書の成長速度を加速することによって圧縮能力を増大できる．そこで，1 文字の入力によって複数のフレーズを構成するなど，フレーズの構成方法によっていくつかの改良案が提案され得る．横尾の符号[6],[7]，Bernstein の Y 符号†† などがその例である．これらの方式ではサイズの小さな入力列に対しても圧縮能力が向上しているが，半面，辞書が飽和する時期も早くなるため，ある程度以上の大きさの入力に対しては LZW 法と圧縮能力にそれほど差がなくなってくる（4.の実験結果を参照）．

2.2 LZ77 法について

文字列の置換による LZ 77 法に基づく圧縮法は，移動窓を用い，先行する同じ文字列の位置と長さを符号として表す．移動窓はリングバッファとして実現されることが多い．この方法では，長い文字列であっても移動窓に存在すれば，先行する文字列を参照する短い符号として表現できる．符号化の部分を工夫することによって高い圧縮能力を実現できる[8]．

しかし逆に，比較的短い置換文字列をあまり効率良く表現できないという問題や，頻繁に出現する文字列であっても，常に同じ符号でしか表せないという問題もある．

これらのアルゴリズムでは一般に，現在の入力と最も長く一致する文字列を移動窓から発見する部分が処理効率を悪化させている．移動窓の大きさを増加させるのに従って圧縮能力も増すが，同時に文字列検索の手間も増大してしまう．このため，辞書の探索を効率的に行えるようなデータ構造やアルゴリズムが多数提案されているが，一般にインクリメンタル分解に基づく圧縮法よりも多くの処理時間を要する．

2.3 二つの方式の融合

上の議論から，LZW 法において辞書にまだ登録されていない文字列が入力中に再び出現した場合に，文字列置換の手法を利用することが考えられる．頻繁に出現する文字列は LZW 符号によって表現し，出現頻

† S. W. Thomas, J. McKie, S. Davies, K. Turkowski, J. A. Woods, and J. W. Orost, *Compress* (version 4.0) 1985.

†† Daniel J. Bernstein: "Y coding", Draft 4b (Mar. 1991). このテキストおよびソフトウェア yabbawhap, version 1.00 (Mar. 1991) はパブリックドメインに置かれている．

度の少ない文字列には文字列置換を適用するのである．

このような折衷的な手法を検討する場合に問題となるのは，符号化の方法も含め，LZW 符号と文字列置換をどのように融合させるかということである．さまざまな考え方があり得るが，本論文では LZW 法が実用的にはほぼ十分な能力をもっていることを考慮し，LZW 法に補助として文字列置換の機能を取り入れる方針とした．

3. 新符号化方式

3.1 LZW 辞書と移動窓

LZW 法のフレーズは既に辞書に登録されている別のフレーズともう 1 文字から構成されるが，以下の議論では，フレーズ A がフレーズ B と文字 c から構成されているとき，フレーズ B はフレーズ A の「親フレーズ」，文字 c はフレーズ A の「付加文字」であると言うことにする．

LZW 辞書と移動窓は次のようにして関係づける．

今，新たなフレーズが LZW 辞書に登録される際に，そのフレーズの付加文字に相当する移動窓上の位置を，フレーズごとに LZW 辞書に登録しておく．また，新たな入力に伴って LZW 辞書が検索された場合にも，そのフレーズの移動窓上の新しい位置を LZW 辞書に登録し直す．但し，移動窓の大きさは有限であるから，移動窓から追い出された文字に関する情報は LZW 辞書からも削除しなければならない．

図 2 は文字列“abaabbab”を処理したときの LZW 辞書と移動窓の関係である．破線の矢印は，そのフレーズと移動窓の古い対応関係を表している．例えば，新しいフレーズ“abb”を登録するときに，既に登録されていたフレーズ“ab”の位置も変更される．

これらの操作は，新たに 1 文字入力するごとに 1 回ずつ行うだけでよい．また，操作の手間は移動窓の大きさには依存しない．

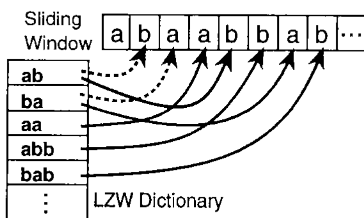


図 2 LZW 辞書と移動窓の対応付け
Fig. 2 LZW dictionary refers to sliding window.

3.2 アルゴリズム

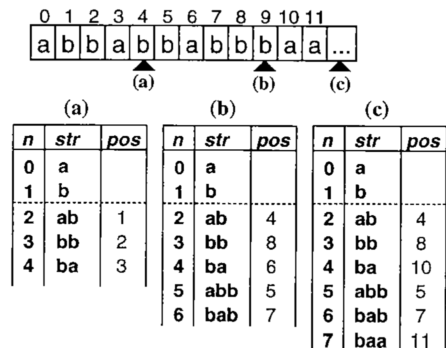
アルゴリズムの概略について述べる．

新しいフレーズを LZW 辞書に登録する際，新しいフレーズの親フレーズが移動窓上の位置情報をもっていたとする．このとき，新しいフレーズの付加文字およびそれ以降の入力文字を移動窓上の文字と順次比較すれば，LZW 辞書には登録されていない置換文字列を移動窓から発見できる．付加文字も含め，2 文字以上の入力文字が移動窓の文字と一致した場合には，親フレーズの LZW 符号に続いて，一致した長さを符号化して出力する．そうでない場合には，通常どおり LZW 符号化の手順を進める．

入力文字列が移動窓の文字列と一致している間，入力文字列は LZW のフレーズに分解を行い，LZW 辞書に登録していくが，出力は行わない．

例として，“abbabbabbbaa”という文字列を符号化する場合について説明する（図 3）．先頭を 0 文字目として 3 文字目まで処理したときの辞書が（a）である．*n* はフレーズに割り当てられた符号，*str* はフレーズ，*pos* はそのフレーズの付加文字の移動窓内の位置である．4 文字目を読み込んだときに構成されるフレーズは“ab”であるが，これは既に辞書に登録されている．そこで，次の文字を読み込み，フレーズ“ab”の *pos* の示す次の位置の文字と比較すると両方“b”である．これ以降，フレーズ“ab”の後ろに引き続く文字と入力文字が同じである間，LZW 法のフレーズを構成し，辞書に登録するが，フレーズの符号は出力しない．また，構成されたフレーズについて *pos* の欄の値も更新する．

（b）は 8 文字目までを処理したときの辞書である．9 文字目は“b”であるが，これは移動窓の対応する文



(a)

n	*str*	*pos*
0	a	
1	b	
2	ab	1
3	bb	2
4	ba	3

(b)

n	*str*	*pos*
0	a	
1	b	
2	ab	4
3	bb	8
4	ba	6
5	abb	5
6	bab	7

(c)

n	*str*	*pos*
0	a	
1	b	
2	ab	4
3	bb	8
4	ba	10
5	abb	5
6	bab	7
7	baa	11

図 3 符号化の例
Fig. 3 An example of coding.

字(6文字目の"a")と一致しない．そこで，ここまで一致した文字数の4を符号化して出力し，これ以降は通常のLZW法と同様の方法に戻る．(c)は11文字目まで処理したときの辞書を示す．10文字目までを表す符号は，0,1,1,2,[4]，4となる．但し，[4]は文字列長を表すとする．文字列長の表現方法については**3.4**で論じる．

図4は符号化法のアルゴリズムを示す．関数NextChar()は1文字入力すると共に移動窓の内容を更新する．変数*ph*は，入力に一致する文字列が移動窓内にある場合，その位置を表すために用い，window[*ph*]はその位置の文字を表す．関数Increment(*ph*)は移動窓の位置*ph*の次の文字の位置を返し，関数OutputLength(*n*)は文字列長*n*を符号化して出力する．また，関数IsChar(*str*)は，符号*str*が文字を表すかどうかを調べる．

```
CompressFile()
{
   Initialize();
   ph = UNUSED; oldph = UNUSED;
   string = NextChar();
   character = NextChar();
   while ( character != END_OF_STREAM ) {
      if ( ph != UNUSED ) {
         ph = Increment( ph );
         if ( window[ph] == character )
            lng = lng + 1;
         else {
            if ( lng > 0 ) {
               OutputLength( lng );
               string = character;
               character = NextChar();
            }
            ph = UNUSED; oldph = UNUSED;
         }
      }
      code = Search( string, character );
      if ( code != UNUSED ) {
         oldph = Position( code );
         string = code;
      }else {
         if ( ph == UNUSED ) {
            OutputBits( string );
            if ( IsChar( string ) )
               oldph = UNUSED;
            else if ( oldph != UNUSED ) {
               ph = Increment( oldph );
               lng = 0;
               if ( window[ph] != character ) {
                  ph = UNUSED; oldph = UNUSED;
               }
            }
         }
         Update( string, character );
         string = character;
      }
      character = NextChar();
   }
   if ( ph != UNUSED && lng > 0)
      OutputLength(lng);
   else OutputBits( string );
}
```

図4 符号化アルゴリズム
Fig. 4 Coding algorithm.

このアルゴリズムは，図1のLZW法のアルゴリズムと比較すると，置換文字列との比較に関する場合分けが行われているため，やや複雑になっている．但しこのアルゴリズムは，LZW法と同様に入力文字数に対して線形時間で実行できる．

復号の際は，符号化のときと同じLZW辞書と移動窓を構築し，LZW符号と置換文字列の長さを用いて入力文字列を復元する．置換文字列の移動窓内の開始位置は，直前のLZW符号と辞書の情報から一意に定まる．従ってLZ77法に基づく多くの圧縮法とは異なり，置換文字列の開始位置を符号化する必要はない．

3.3 復号化について

3.2で述べたように，本方式では置換文字列に対してもLZWのフレーズへの分解と辞書への登録を行う．置換文字列にこの操作を行わない場合，圧縮能力が悪化することが実験の結果から判明している．従って，復号化の処理においても，得られた置換文字列をLZWのフレーズに分解し，辞書に登録するという操作が必要になる．

LZW法は復号化の際に，あるフレーズと文字の組合せが既に辞書に登録されているかを検索して調べる必要がなく，このことが復号化の高速化をもたらして

```
ExpandFile()
{
   Initialize();
   string = InputBits();
   PutChar( string );
   character = string;
   newcode = InputBits();
   while ( newcode != END_OF_STREAM ) {
      ph = Position( string );
      UpdatePosition( string );
      if ( IsLength( newcode ) ) {
         lng = GetLength( newcode );
         while (lng > 0) {
            lng = lng - 1;
            ph = Increment( ph );
            ch = window[ph];
            code = Search( string, ch );
            if (code != UNUSED) string = code;
            else {
               Update( string, ch );
               string = ch;
            }
            PutChar( ch );
         }
         newcode = InputBits();
         if (newcode == END_OF_STREAM) return;
         character = DecodeString( newcode );
      } else {
         character = DecodeString( newcode );
         Update( string, character );
      }
      string = newcode;
      newcode = InputBits();
   }
}
```

図5 復号化アルゴリズム
Fig. 5 Decoding algorithm.

いる．これに対し，本方式では復号化の際にも符号化の際と同様な LZW 辞書と移動窓を再構成する必要がある．なお，横尾の符号をはじめ，LZW 法を改良した方式には復号化の際に LZW 辞書を再構成する必要があるものが多い．

図5に復号化のアルゴリズムを示す．この操作も入力サイズに対する線形時間で行える．

関数 Position(*str*) は符号 *str* に対応するフレーズの移動窓内の位置を辞書から得る．関数 UpdatePosition(*str*)は，符号 *str* を文字列に分解すると共に移動窓と LZW 辞書の内容を更新する．関数 IsLength (*str*)は，符号 *str* が文字列長を表すかどうか調べ，関数 GetLength(*str*) はその長さを値とする．関数 DecodeString(*str*) は符号 *str* に対応する文字列を書き出す．

3.4 文字列長の表現

図4に示したアルゴリズムでは，LZW 符号と置換文字列の長さを具体的にどのように符号化して出力するかに関しては記述していない．この実現にはさまざまな方法が考えられるが，今回実験で作成したプログラムでは以下の二つの方法を用いた．

方法(a)　compress では LZW 符号は当初9ビットで表され，符号が大きくなるに従ってビット数を増加させている．この方法では，符号を表すために使用されているビット数が n であれば，表現できる最大の数 M は 2^n-1 である．その時点で最大の LZW 符号に相当する整数値が Z ならば，$(Z+1)$から M までの整数は使用されない．置換文字列の長さ L が $L<M-Z$ の場合，$(M-L)$ は n ビットで表現され，LZW 符号と区別できる．そこで，LZW 符号は compress と同じ方法で表現し，置換文字列の長さは $(M-L)$ を出力することにする．但し，$L \geq M-Z$ の場合には，ほかの LZW 符号と区別できる符号をあらかじめ用意しておき，この符号の直後に L を出力する．実験の結果では大半の場合について前者の方法で符号化できている．

方法(b)　置換文字列は，短いものほど高い頻度で出現する．そこで，これらの文字列長の平均よりも十分大きな数 S（以下の実験では $S=128$ とした）を決め，1から S までの置換文字列の長さを表すために S 個の特別の符号をあらかじめ LZW 辞書に含めておく．文字列長 L が S より長い場合には，もう一つ特別に用意した符号の直後に L を出力する．方法(a)は LZW 法の符号化方式にもともと存在していた冗長な部分を利用していることになるが，この方法(b)ではその冗長さを残している．また，初期状態の LZW 辞書のサイズが大きくなるため，方法(a)よりは圧縮率が悪い．

以下の実験では，特に断らない限り方法(a)を用いて性能評価を行う．但し，**4.5** で方法(a)と方法(b)について比較実験を行い，置換文字列の長さを符号化することが LZW 法の改良になっていることを示す．

4. 実験による性能評価

4.1 比較対象とした圧縮法について

本論文の目的は，提案したアルゴリズムが LZW 法の圧縮能力を改善していることを示すことである．そこで，LZW 法とその改良である横尾の符号，および Bernstein の Y 符号を比較対象として取り上げる．また，LZ 77 法に基づく文字列置換による圧縮法の代表として LZSS (Lempel-Ziv-Storer-Szymanski) 法[9] を取り上げ，圧縮の傾向などに関して比較検討を試みる．

ここで，LZW 法，LZSS 法のプログラムとして，文献(2)に掲げられたプログラムを使用した．また，本論文のアルゴリズムによるプログラムおよび横尾の符号（文献(6)でのアルゴリズム1）のプログラムはこの LZW 法のプログラムに手を加えて実現している．LZW 法，横尾の符号，および本方式のいずれのプログラムも，フレーズを表す符号があらかじめ定めた最大ビット数で表現できなくなった時点で辞書を初期化する方式を用いている．Y 符号のプログラムは Bernstein がネットワーク上に公開しているプログラム yabba を用いた．

以下の説明および図表では，最大ビット長が 14,16 の LZW 法をそれぞれ LZW 14, LZW 16 と呼ぶ．横尾の符号も同様に Yok 14, Yok 16 とする．yabba では使用するメモリのサイズを指定できるが，ここでは 16381 と 65533 を用い，それぞれ yab 16, yab 65 と呼ぶ．これらの辞書に登録できるフレーズ数はそれぞれ LZW 14, LZW 16 に対応する．LZSS 法ではリングバッファのサイズを 2 kByte, 8 kByte とし，それぞれ LZSS 2 k, LZSS 8 k と呼ぶ．それぞれの先読みバッファ長は 17 Byte, 18 Byte である．本方式のプログラムは LZW 辞書の最大ビット数とリングバッファの長さの二つのパラメータをもつが，LZW 辞書の最大ビット長が 14 でリングバッファ長が 2 kByte のプログラムを O 14-2 k のように呼ぶ．

4.2 LZW 辞書と移動窓の大きさについて

図6は，本方式について LZW 辞書の最大ビット数

と移動窓の大きさを変化させたときの圧縮率の変化を調べたものである．比較のためにLZW法の結果を加えている．ここで，圧縮率＝(圧縮後のサイズ/入力サイズ)である．(a)のデータはUNIX上のテキストエディタであるemacsのソースプログラムおよび文書類をUNIXの*tar*コマンドで一つのファイルにまとめたもの(10.0 MByte)，(b)のデータは英語版聖書[†](4.6 MByte)である．

LZW辞書のサイズが小さい場合に，移動窓を付加する効果が大きいことがわかる．

バッファ長を大きくとっても，ある程度以上の大きさになると，効果があまり現れないということもわかる．これには以下の二つの理由が挙げられる．

提案したアルゴリズムでは，LZW辞書から移動窓への位置情報は，常に一番最近出現したフレーズを参照するように更新される．これは，移動窓の更新に伴って位置情報が無効になるのをできるだけ防ぐためである．この結果，LZW辞書からは移動窓の新しい入力の部分をより多く参照するようになり，あまり長いバッファを使っても効果が現れにくくなると考えられる．

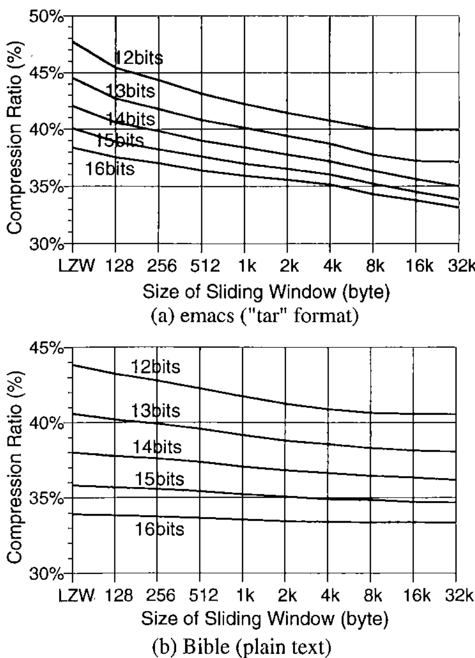


図 6 移動窓のサイズと圧縮率の変化
Fig. 6 Relation of sliding window size to compression ratio.

また，LZW辞書のサイズに比べてバッファ長が大きい場合，移動窓の長さ分の情報を利用する前にLZW辞書が飽和してしまうことがある．図6で最大ビット数が12の場合，移動窓の大きさを8 kByteより増やしても圧縮率の改善が見られないのはこのためであると考えられる．

4.3 さまざまなデータによる比較

図7は文献(1)で用いられている比較用データ[††]に対する処理結果である．それぞれのデータの概要を表1に示す．

移動窓の効果がはっきり現れるデータとそうでないデータがあるが，一般にLZSS法の圧縮率が良い場合に移動窓の効果も高くなる傾向がある．特にソースプログラムなどについて効果が著しい．移動窓の効果があまり出ていない場合でも，同じサイズの辞書をもつLZW法と同程度の結果を出している．

また，その他の方法と比較した場合，データによる差異はあるものの，これらの中で最悪という例はない．データの性質に左右されず，常に比較的良い圧縮率を示している．

4.4 入力長の変化による実験

図7の結果からは，さまざまな入力長のデータに対して圧縮率がどのように変化するかを知ることができない．そこで，次の方法で圧縮率の変化を測定した．データとしては4.2と同様，emacsのソースプログラムと聖書を用いる．emacsファイルについては先頭か

表1 比較用データの内容

データ名	内　容	大きさ (Kbyte)
bib	文献目録(refer形式)	111.3
book1	小説	768.8
book2	論文(troff形式)	610.9
geo	バイナリデータ	102.4
news	電子ニュース	377.1
obj1	VAX実行形式	21.5
obj2	Macintosh実行形式	246.8
paper1	論文(troff形式)	53.2
paper2	論文(troff形式)	82.2
pic	ビットマップ	513.2
progc	Cプログラム	39.6
progl	Lispプログラム	71.6
progp	Pascalプログラム	49.4
trans	画面操作シーケンス	93.7

† The King James' Bible, Project Gutenberg 2nd Version (1992).
†† Data Compression Corpus. このデータは最近，圧縮プログラムの性能を評価するための標準的なデータとして用いられている．ネットワーク経由で入手可能である．

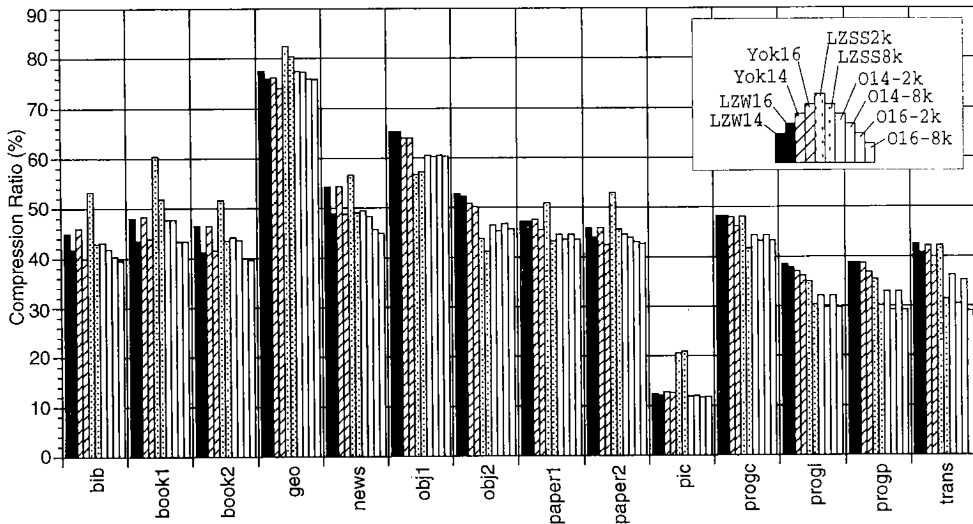


図 7 さまざまなデータによる比較
Fig. 7 Comparison using various data.

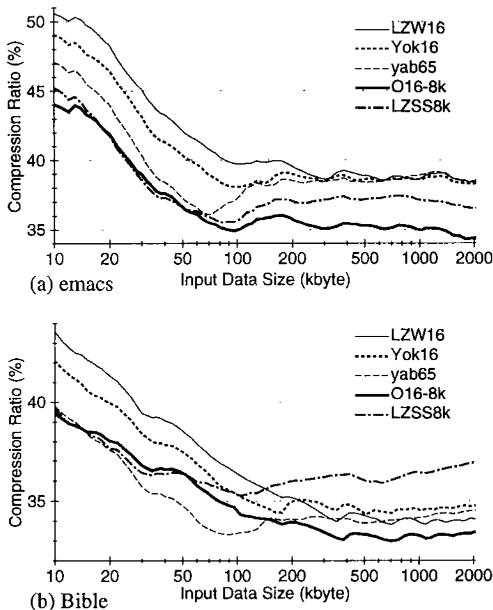


図 8 入力長による圧縮率の変化
Fig. 8 Relation of input data size to compression ratio.

ら 0,2,4,6,8 MByte, 聖書ファイルについては先頭から 0, 0.5, 1.0, 1.5, 2.0 MByte の位置から始まる大きさ 2 MByte のファイルを用意する．これら五つのファイルのそれぞれについて，入力長が増加するのに伴う圧縮率の変化を計測し，結果の平均を求めた．このようにすれば，ファイルの特定の部分に依存するデータの変動を少なくし，圧縮法の特性を明らかにできる．

図 8 (a) は emacs ファイル，(b) は聖書ファイルについて，入力長の変化に伴う各圧縮法の圧縮率の推移の傾向を表す．本方式は入力サイズが小さいうちから圧縮能力が高い．また，横尾の符号および Y 符号は，入力サイズが小さい場合には LZW よりも高い圧縮能力を示すが，入力サイズが大きくなって辞書が飽和するようになると LZW とほぼ同等の圧縮能力になるという点も注目される．これらの例に見られるように，移動窓の効果が大きいデータに対しては，本方式は LZSS 法と似た特性を示し，移動窓の効果があまり出ないデータに対して LZW 法と似た特性を示す．

4.5 文字列長の符号化方法の比較

これまでの実験では，置換文字列の長さの符号化に **3.4** で述べた方法 (a) を用いていた．方法 (a) は LZW 法の符号化方式にもともと存在していた冗長な部分を利用しているため，厳密には置換文字列の長さを符号化することによって圧縮率を改善しているとは断言できない．そこで，この冗長な部分を利用しない方法 (b) との比較を行った．なお，方法 (b) で文字列長を表すためにあらかじめ LZW 辞書に含める符号の数 S は 128 とした．

図 9 は，LZW 辞書の最大ビット数が 12,14,16 の場合について移動窓の大きさを変化させ，対応する LZW 法の結果に対する大きさの割合を示したもので

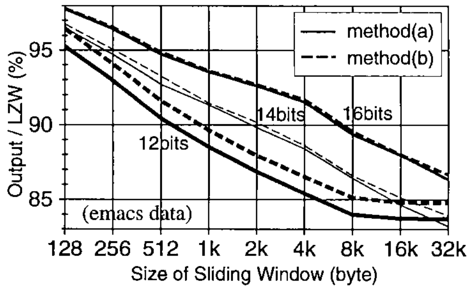


図 9 二つの符号化方式の比較：移動窓のサイズ
Fig. 9 Comparison of coding methods: size of sliding windows.

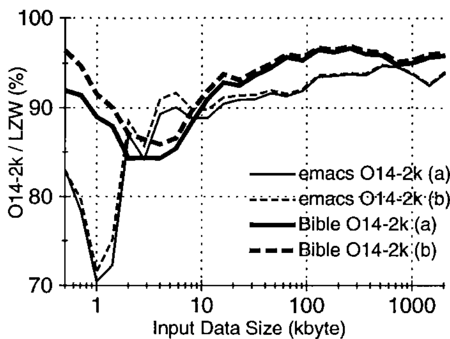


図 10 二つの符号化方式の比較：入力サイズの変化
Fig. 10 Comparison of coding methods: input data size.

ある．データとしては emacs ファイルを用いた．図中の実線が方法(a)，破線が方法(b)である．

図 10 は O 14-2 k を使って，emacs データ，聖書データの先頭部分 2 MByte までのさまざまな大きさのデータを圧縮した結果と，対応する LZW 14 の結果の割合を表している．

これらの実験から，LZW 法の冗長な符号化方式をそのまま残した方法(b)でも LZW 法の結果を改良していることがわかる．このことは，提案アルゴリズムが LZW 符号の代わりに置換文字列の長さを符号化することによって LZW 法を改良していることを示すものである．

辞書の最大ビット数が小さい場合，および入力データが小さい場合に方法(b)の圧縮率が悪いのは，LZW 辞書の初期サイズの大きさが影響しやすいためと考えられる．なお，方法(b)は，長法(a)との比較のためのものであり，$S=128$ という設定にも特に理論的な根拠はない．また，その改良にはあまり意義を見出せないが，長い置換文字列の出現に従って長さを表す符号を動的に辞書に登録するなどの方法を採用すれば，上記の問題点は解決可能であると思われる．

表 2 処理時間の比較

	圧縮	復元
LZW16	53.1(3.1)	31.2(3.7)
O16-2k	67.6(3.3)	51.8(3.8)
O16-8k	66.7(3.1)	51.4(3.7)
Yok16	77.7(3.2)	68.3(3.9)
LZSS8k	345.6(4.2)	26.2(3.5)

各欄左はuser time(秒)，括弧内はsystem time(秒)である．

4.6 実行時間

4.1 で述べたように，LZW 法，LZSS 法，横尾の符号，および本アルゴリズムはすべて文献(2)のプログラムに基づいており，入出力部分は同一のルーチンを使用している．これらのプログラムが聖書ファイルを圧縮，および復元するのに要した実行時間を表 2 に示す．このデータは NeXTstation Turbo 上で 5 回計測したものの平均である．

本方式のプログラムが圧縮に要する時間は LZW 法に比べ，25%程度の増加にとどまっていることがわかる．但し，復元に要する時間は 60%程度増加してしまう．これは，3.3 で述べたように，LZW 辞書の再構成とフレーズの検索が必要なためと考えられる．また，移動窓の大きさを変えても，実行時間があまり変化しないこともわかる．移動窓が大きい方がわずかに速いのは，圧縮率の違いによって入出力データ量が減少したためと考えられる．

5. 議　　論

5.1 置換文字列の検索

本方式では，LZW 法がフレーズとして切り出した文字列のみを置換文字列の開始端の候補としている．また，その候補はまったくないか，ある場合でも一つだけという方法をとっている．しかしこの方法で，現在の入力列と最も長く一致する置換文字列を移動窓から検索できるわけではない．このことは，LZW 法が入力列からフレーズを取り出すとき，そのフレーズの途中から始まる文字列がフレーズとして辞書に登録されないことから明らかである．

移動窓の中から，現在の入力列と最も長く一致する文字列を検索するという方法は LZSS 法などと同様，検索にかなり手間がかかる．最良の置換文字列ではなくても，複数の候補からより長く一致する置換文字列を選択するという方式もいくつか考えられる．しかし

これらの場合には，置換文字列が移動窓のどこにあるかという情報も符号として出力に含めなければならなくなるため，圧縮能力が向上するかどうかは疑問である．

5.2 LZW 法の改良アルゴリズムへの応用

前述したように，横尾の符号，Y 符号などでは LZW 法のフレーズの部分文字列を利用して新しいフレーズを生成する．これらに本方式と同様の移動窓を付加することも考えられよう．こうすれば，一定の大きさしかない移動窓の中から置換文字列を見つける可能性を高めることができるであろう．しかし反面，アルゴリズムの複雑化，処理時間の増加などが予想される．また，単に移動窓を大きくとった場合よりも有効に働くかどうかは不明である．この問題に関しては引き続き検討が必要である．

5.3 連続する同一パターンの圧縮

本方式では，入力文字列と置換文字列の比較をリングバッファ上で行う．このため，同じ文字，あるいは文字列が繰り返される場合には，入力文字が次々に置換文字列として使われるという現象が見られ，圧縮効率を非常に高める効果がある．図 3 の例は，文字列 “abb”が連続する場合の例にもなっている．

同様な現象は LZSS 法などでも見られるが，LZSS 法では先読みバッファの長さ以上の文字列は符号化できないという制限がある．これに対して本方式は，リングバッファの長さにも関係なく，理論的にはパターンがいくら長く連続しても非常に短い符号で表現することができる．この特徴から，本方式は同一のパターンが連続して現れるデータの圧縮に効果があるものと考えられる．

表 3 に例を示す．この実験で用いたデータは文字列 “yes”と改行文字の 4 文字の連続のみからなる．

5.4 LZW 法と LZ 77 法の融合について

インクリメンタル分解と文字列置換の融合による圧縮法としては，本論文のほかに横尾[10] の提案がある．横尾は，文献(7)の Algorithm 4，あるいは LZFG 法[8] を中心に議論を行っているが，インクリメンタル分解によるフレーズを表す符号の代わりに置換文字列の長さを符号化するという基本的なアイデアは本論文と同じである．

しかし，文献(10)では移動窓を用いる方法に関しては論じられていない．本論文は，LZW 法と LZ 77 法の融合に移動窓を利用する具体的かつ実用的な実現の一つであると言える．また，本論文では各種の実験を通して，LZW 法と LZ 77 法の融合方式が LZW 法と LZ 77 法の特徴を併せもった圧縮特性を示すことを明らかにした．

今，x, y が文字，X, Y が文字列を表し，また，Xy が文字列 X に文字 y を連接した文字列を表すものとする．LZW 法では，Xy がフレーズとして辞書に登録されている場合，X も必ず辞書に保存されている（条件 1）．文献(10)ではこれに加え，xY がフレーズとして辞書に登録されている場合には Y も辞書に保存されていること（条件 2）を，融合方式のアルゴリズムが満たすべき条件として挙げている．これに対し，本論文で提案した方法は LZW 辞書をもつことを仮定しているだけであり，上記の条件 2 は必須ではない．このため，辞書の構築を高速に行うことが可能である．

条件 2 が不要である理由としては，本方式が過去の入力系列を記憶するための手段として移動窓を用いていることと，5.1 で論じたように，あるフレーズが参照する移動窓上の文字列位置を一つに限定していることが挙げられる．この意味において，本論文は横尾の提案とやや異なるアプローチによる融合方式のクラスが存在することを示すものと考えられよう．

表 3 連続する文字列の圧縮

Original	512000	1024000
LZW16	2688	3946
Yok16	2185	3180
LZSS8k	64008	128007
O16-8k	16	16

(byte)

6. むすび

LZW 法に移動窓を付加し，文字列置換による符号化を組み合わせる新符号化法を提案した．また，さまざまなデータに対して適用実験を行った結果から，本方式は LZW 法の圧縮能力を改善すると共に，処理速度の面からも実用性の高い方式であることを示した．

今後は，5.で述べたように，他の LZW 法の改良アルゴリズムへの適用実験，アルゴリズムの高速化，および融合方式に関するいっそうの考察が課題である．本論文ではアルゴリズムと実験結果の報告を行ったが，本方式および融合方式に関しては，その性質をより体系的に明らかにする必要があろう．

謝辞　本研究を行う上で，常に励まして頂いた大阪大学情報処理教育センターの福岡秀和教授，松浦敏雄助教授に深く感謝致します．

文献

(1) Bell T. C., Cleary J. G. and Witten I. H.: "Text Compression", Prentice Hall (1990).

(2) Nelson M.: "The Data Compression Book", Prentice Hall (1991).

(3) Ziv J. and Lempel A.: "Compression of Individual Sequences via Variable-Rate Coding", IEEE Trans. Inf. Theory, **IT-24**, 5, pp. 530-536 (Sept. 1978).

(4) Welch T. A.: "A Technique for High-Performance Data Compression", IEEE Computer, **17**, 6, pp. 8-19 (June 1984).

(5) Ziv J. and Lempel A.: "A Universal Algorithm for Sequential Data Compression", IEEE Trans. Theory, **IT-23**, 3, pp. 337-343 (May 1977).

(6) 横尾英俊："実時間パターン照合によるデータ圧縮の高性能実用算法", 情報処理学会論文誌, **30**, 10, pp. 1309-1315(1989-10).

(7) Yokoo H.: "Improved Variations Relating the Ziv-Lempel and Welch-Type Algorithms for Sequencial Data Compression", IEEE Trans. Inf. Theory, **38**, 1, pp. 73-81 (Jan. 1992).

(8) Fiala E. R. and Greene D. H.: "Data Compression with Finite Windows", Commun. ACM, **32**, 4, pp. 490-505 (April 1989).

(9) Storer J. A. and Szymanski T. G.: "Data Compression via Textual Substitution", J. ACM, **29**, 4, pp. 928-951 (Oct. 1982).

(10) 横尾英俊："Ziv-Lempel 符号の融合方式によるデータ圧縮", 信学技報, **IT92**-135(1993-03).

(平成5年2月24日受付，6月16日再受付)

荻原　剛志

昭60山梨大・工・計算機科学卒．昭62同大大学院修士課程了．平2阪大大学院博士課程了．同年同大・情報処理教育センター助手．ソフトウェア工学，データ圧縮などの研究に従事．工博．情報処理学会，日本ソフトウェア科学会各会員．

飯田　元

昭63阪大・基礎工・情報卒．平2同大大学院博士前期課程了．同年同後期課程入学．平3同大・基礎工・情報・助手．ソフトウェア開発プロセスおよび開発支援環境の研究に従事．日本ソフトウェア科学会会員．

井上　克郎

昭54阪大・基礎工・情報卒．昭59同大大学院博士課程了．同年同大・基礎工・情報・講師．昭59〜61ハワイ大学助教授．平3阪大・基礎工・情報・助教授．ソフトウェアプロセス，関数型言語の処理系等の研究に従事．工博．情報処理学会，ACM, IEEE各会員．

鳥居　宏次

昭37阪大・工・通信卒．昭42同大大学院博士課程了．同年電気試験所（現電子技術総合研究所）入所．昭50ソフトウェア部言語処理研究室室長．昭59阪大・基礎工・情報・教授．平3奈良先端大・情報・教授（阪大併任），図書館長．工博．ソフトウェアメトリクスやフォーマルな開発プロセスなど，定量的取扱いを中心とするソフトウェア工学に興味をもつ．IEEE Transaction on Software Engineering および IEEE Software 誌などの編集委員．IEEE，ACM，情報処理学会各会員．