アルゴリズム 32－2
（1993. 3. 18）

適応型データ圧縮アルゴリズムの設計と評価

小島 俊輔，中嶋 卓雄，中村 良三
熊本大学 工学部

本論文では，適応型データ圧縮アルゴリズム LZSS 法に LZW 法のフレーズの概念を取り入れた新たなデータ圧縮アルゴリズムを提案する．

提案するデータ圧縮アルゴリズムは，高圧縮が可能な LZSS 法において符号化済み文字列が格納されているバッファをフレーズとして効率的に利用することにより，実行速度をあまり低下させることなく，より高圧縮率を実現している．このアルゴリズムを各種のドキュメントファイルに適用した結果，従来の LZSS アルゴリズムと比較して約 4～5%ほど圧縮率を高めることができた．

Design and Evaluation of a New Adaptive Data Compression Algorithm

Shunsuke Oshima, Takuo Nakashima, Ryozo Nakamura

Faculty of Engineering, Kumamoto University
2-39-1 Kurokami, Kumamoto-shi, 860, Japan

This paper describes a new adaptive data compression algorithm based on LZSS technique. The algorithm partially splits the already encoded buffer by phrase, and could obtain the better compression ratio utilizing the phrase dictionary. The proposed compression technique used has evaluated on textual data such as documents and source codes. We have achieved better compression ratio than the conventional techniques.

1 はじめに

近年，計算機で扱われるデータ量は急速に増加しており，記憶コストや通信コストを削減するためのデータ圧縮化技法は各種の分野において必要不可欠の技法となりつつある．特にテキストデータの圧縮は最も一般的な課題である．

テキストデータ圧縮アルゴリズムは，各文字を統計的に評価し，その出現確率によってコードを生成する統計的方式 (statistical technique) と，一連の文字列を辞書のインデックスと置き換えることによって圧縮を行なう辞書方式 (dictionary technique) に分類される [7].

統計的方式は出現確率の大きいアルファベットには短い符号 (code) を，出現確率の小さいアルファベットには長い符号を割り当てることによって，平均符号長を短くする．統計的方式による圧縮アルゴリズムは符号化を施す前からソースメッセージの種類が決まっていることから，既定語方式 (defined-word schemes) と呼ばれることもある．一般に知られたシャノン・ファノ符号化 (Shannon-Fano coding)，ハフマン符号化 (Huffman coding)，算術符号化 (Arithmetic coding) などはこの方式に該当する [8].

辞書方式は情報源に内在する文脈 (context)，すなわち連続した文字列を切りだし，その文字列を辞書に保存し，以降同じ文字列が出現したときには，辞書のインデックスと置き換えることによって符号長を短くする．したがって，この方式はマクロコーディング (macro coding) とかコード帳による方式 (codebook technique) とも呼ばれる．また，この方式はソースメッセージを可変の文字列に分割することから自由解析法 (free parse) とも呼ばれている．テキストデータには重複する文字列が数多く存在し，さらにインデックスの方が符号長も短くなることから，一般的に統計的方式より辞書方式が用いられる場合が多い．

さらに，この辞書方式は，メッセージ集合から符号語集合への写像が伝送開始前から固定されている静的 (static) 手法と，写像が時間に応じて変化する動的 (dynamic) 手法とに分類される．動的手法は時間によるソースメッセージの特性変化に適応できるという意味で適応型 (adaptive) とも呼ばれている．適応型は情報源のパターンの変化に適応できることから実時間でのデータ圧縮が可能である．おり，特に Lempel と Ziv が提案した適応型 LZ アルゴリズム [1] は実用的な各種のデータ圧縮アルゴリズムの基本となっている．

本稿では，適応型データ圧縮アルゴリズム LZSS 法に LZW 法のフレーズの概念を取り入れた新たなデータ圧縮アルゴリズムを提案する．

まず，2 章では，LZ アルゴリズムとそれを発展させた LZW と LZSS の特徴について述べる．3 章では，提案する適応型データ圧縮アルゴリズムについて述べ，4 章では，提案するアルゴリズムと他の LZ アルゴリズムとの圧縮率などを評価し，比較検討する．

2 LZ アルゴリズム

LZ では，すでに符号化した文字列を格納するバッファを設け，現在符号化しようとする文字列の長さが最大となるような文字列をバッファ中から検索し，バッファ中のマッチングした文字列の位置 m と文字列の長さ l の組 (m,l) によって符号化される．

たとえば，図 1 に示すように [7]，LZ アルゴリズムでは文字列 “abbaabbbabab” は “abba(1,3)(3,2)(8,3)” と符号化される．“abba” までは，1 文字のマッチングしか成功せず，符号化してもビット数が多くなるので，そのまま出力され，次の “abb” から，位置 1，長さ 3 の文字列とマッチングが成功し，(1,3) と符号化され，“babab” も同様に (3,2)(8,3) と符号化される．

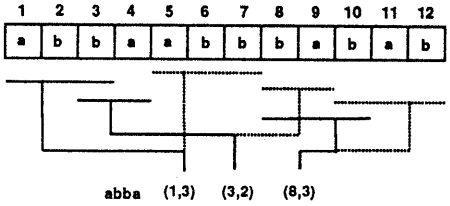

図 1 LZ アルゴリズムによる圧縮例

LZ では，すでに符号化した文字列を格納するバッファに固定長のウィンドウ (fixed-size window) を用いるのか，すべての文字列を格納する成長するウィンドウ (growing window) を用いるのか，2 通りのアプローチがある．

成長するウィンドウを使うと，マッチングを試み

るパターンが多くなり，圧縮率が向上する可能性があるが，ウィンドウが大きくなり過ぎると計算時間が増加し，さらにバッファの位置を示す符号長も大きくなり圧縮率が低下する．一方，固定長のウィンドウを使うと，符号化される符号長が小さくなり，符号化処理も高速化され，これらの問題は改善されるが，マッチングを試みるパターンが少なくなり，圧縮率が低下する可能性がある．

このように，LZ における符号長に直接反映するデータ構造は慎重に決定する必要がある．

多くの LZ 型圧縮アルゴリズムは，Ziv と Lempel によって 1977 年および 1978 年に発表された，LZ77[1] および LZ78[2] と呼ばれる 2 つの技法に分類できる．以下では，LZ77 に基づく LZSS[3] と LZ78 に基づく LZW[4] について述べる．

2.1 LZSS アルゴリズム

LZ77 では，図 2に示すように，文字列を格納するバッファとして固定長（N文字）のスライディングウィンドウ (sliding window) を用いている．スライディングウィンドウは，符号化前の文字列を格納するサイズ Fの先読みバッファ(lookahead buffer) とサイズ $N-F$の符号化済み (already encoded) バッファから構成される．

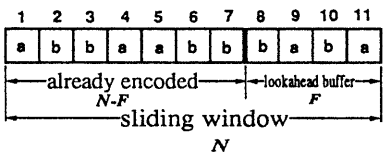

図 2 LZ77 におけるウィンドウ構成

符号化は，この先読みバッファ中の最長の文字列と一致する文字列を符号化済みバッファから探索し，先読みバッファと符号化済みバッファの境界から，一致した位置までのオフセット i，一致文字列の長さ j，および先読みバッファ中の一致しなかった最初の文字 a の 3 つの組$< i,j,a >$によって表現される．

たとえば，バッファのサイズが $N=11, F=4$ の場合，文字列 “abcabcbacbababca ...” を符号化するとき，まず，$N-F=11-4=7$ 文字の符号済みバッファには，空白が入れられ，先読みバッファには “abca” が入れられる．符号化が開始すると，図 3に示すように，先頭の “a”，“b”，“c” は最初に現れる文字なのでそれぞれ，$<?,0,a>$, $<?,0,b>$, $<?,0,c>$ と符号化される．ここで，不定の値を’?’ と表している．次の “abcb” は$< 3,3,b >$，“ac” は$< 4,1,c >$，“bab” が$< 3,2,b >$と順次符号化される．

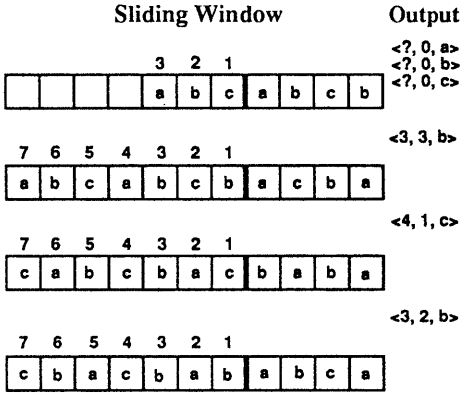

図 3 LZ77 におけるスライディングウィンドウの例

図 3に示した例からも明らかなように，LZ77 で符号化される 3 つ組には，単に 1 文字を表す符号も含まれており，全体の符号長を増加させる要因となっている．

Storer と Szymanski が提案した LZSS アルゴリズム [3] は 1 つの 3 つ組の符号長より短く符号化できる文字列はそのまま出力させ，符号長の増加を防いでいる．符号はオフセット i と一致文字列長 jの 2 つの要素からなる 2 つ組$< i,j >$とし，ポインタと呼ばれている．ここで，一致文字列長を $length$，直接符号化したほうが符号長が短くなる文字列の長さを p とすると，LZSS のアルゴリズムは図 4のように表される．

たとえば，図 3に示した例について，$p=1$ の場合の LZSS による符号化を図 5に示す．まず，$N-F=7$ 文字の符号済み部分には，空白が入れられ，先読みバッファには “abca” が入れられる．符号化が開始すると，先頭の “a”，“b”，“c” はそのまま符号化され，次の “abc” は$< 3,3 >$，さらに “b”,“a” はそのまま符号化され，“cba” は$< 3,3 >$と順次符号化される．

LZSS では [6]，符号の中にポインタと文字が混在するため，1 ビットのフラグを利用して識別している．オフセットは $\lceil \log_2 N \rceil$ ビット，一致する最大文

```
while 先読みバッファが空でない do
  符号化済みバッファの中から先読みバッファと
  最も長く一致する文字列へのポインタ
  <offset, length>を求める
  if length > p then
    ポインタ<offset, length>を出力
    ウィンドウを length 文字シフト
  else
    先読みバッファの先頭の 1 文字を出力
    ウィンドウを 1 文字シフト
```

図 4 LZSS の符号化アルゴリズム

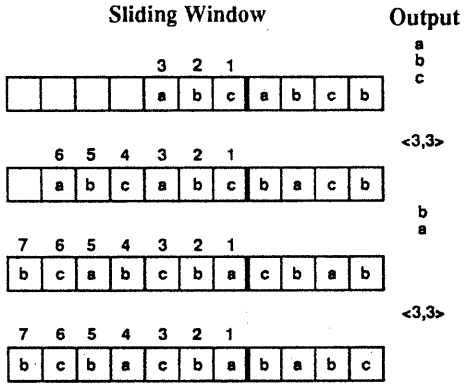

図 5 LZSS の符号化よるスライディングウィンドウの例

字列の長さは p より大きくなり，$\lceil \log_2(F-p) \rceil$ ビット要するので，符号長 L（ビット）は，

$$L = 1 + \lceil \log_2 N \rceil + \lceil \log_2(F-p) \rceil \qquad (1)$$

と表される．

2.2 LZW アルゴリズム

LZ77 では，すでに符号化されマッチングの対象となる文字列は 1 文字ずつ区切られていると考えたが，LZ78 では，フレーズ (phrase) と呼ばれる文字列によって，ウィンドウを区切りマッチングする文字列の組を限定している．

LZ78 においては，フレーズはすでに登録されたフレーズに新しい文字を付け加えることによって構成される．したがって，重複するフレーズは存在せず，圧縮率を向上させている．符号はフレーズ i とそれに続く文字 a から構成される 2 つの組 (i, a) によって表される．

たとえば，文字列 “aaabbabaabaaabab” は図 6に示すように 7 つのフレーズに分割される．まず “a” が $(0, a)$ と符号化され，“aa” が 1 つ前のフレーズ 1 に “a” が続く文字列なので $(1, a)$ と符号化される．続く文字列もすでに出現したフレーズとのマッチングを繰り返し同様に符号化される．

Input:	a	aa	b	ba	baa	baaa	bab
Phrase number:	1	2	3	4	5	6	7
Output:	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

図 6 LZ78 による符号化の例

LZSS においては，LZ77 の符号に含まれている 1 文字からなる符号を直接文字として出力させ，圧縮率を向上させたのと同様に，LZW では，LZ78 の符号に含まれている 1 文字からなる符号を，辞書に登録しておき，辞書のインデックスだけで符号を表現している．

たとえば，文字列 “aababababaaa” の LZW による符号化を，図 7 に示す．まず，入力されるメッセージに含まれる可能性のあるすべての文字，ここでは “a” と “b” を辞書に登録する．次に，順次，文字列を読み込みながら符号化する．まず，“a” を読み込むと，すでに辞書に登録されているので，パターンマッチを試みる時の一時的な変数ωに “a” を代入する．出力は次の文字を読み込みマッチング処理を行なうまで遅延される．次に “a” を読み込み，ωa が辞書に存在するかどうか調べ，存在しないため，1 文字目の “a” の符号である辞書のインデックス 0 を出力し，ωa である “aa” を辞書に登録し，$a \rightarrow \omega$ とする．次に “b” を読み込み，ωb が辞書に存在しないので，同様に 0 を出力し，“ab” を辞書に登録し，$b \rightarrow \omega$ とする．このように辞書に登録しながらインデックスだけの符号を出力する．

LZW はフレーズだけのマッチングによる高速性から，UNIX の標準的な圧縮プログラムである compress[5] にも採用されている．

2.3 LZSS と LZW の比較

前述したように，LZSS では固定長のスライディングバッファを用いているにもかかわらず，1 文字

Input: a a b ab aba aa

Output: 0 0 1 3 5 2

Phrase list:

Phrase number	Phrase	Derived from phrase
0	a	initial
1	b	
2	aa	0 a
3	ab	0 b
4	ba	1 a
5	aba	3 a
6	abaa	5 a

図 7 LZW による符号化の例

単位にマッチングを試みているので，高圧縮率が実現できている．一方，LZW では，フレーズを導入して一致するパターン数を減少させてアルゴリズムを高速化している．さらに圧縮率を維持するために，成長するウィンドウを用いて，一致するパターンを徐々に増加させている．

3 提案する圧縮アルゴリズム

LZSS 法には次のような問題点が存在する．LZSS において符号化済みバッファに注目すると，先読みバッファの先頭の文字列が符号化されウィンドウがスライドしたとき，符号化済みバッファには同じパターンの文字列が複数個存在することになる．このような重複が繰り返し発生すると，圧縮率が低下する．さらに，先読みバッファに注目すると，符号の中に一致文字列の長さが含まれているので，マッチングする文字列の最大長を表す Fのサイズも大きくできず，圧縮率が向上しない原因となっている．

3.1 アルゴリズムの特徴

LZSS における問題点を克服するため，次のような特徴を持つ圧縮アルゴリズムを考案した．

1. 符号化処理の高速性をできるだけ保持したまま，圧縮率を高めるため，LZSS 法に LZW で用いられている，フレーズの概念を導入する．
2. フレーズを新たに構成した辞書に登録することにより，符号化済みバッファにはその辞書のインデックスのみ登録する．
3. 長い文字列とのマッチングも可能にするため，先読みバッファの長さを可変とする．
4. 一致文字列の長さを符号として出力するのではなく，符号化済みバッファにおける一致文字列を示す 2 つのインデックスの差分を符号として出力する．

3.2 データ構造

前述した特徴を有効に実現するため，次のようなデータ構造を用いる．

辞書

一度符号化された文字列はフレーズとして辞書に登録する．符号化済みバッファにはその辞書のインデックスを格納する．マッチングは符号化済みバッファの文字を 1 文字ずつ調べるが，フレーズとして辞書に登録した部分は，文字単位に調べるのではなくフレーズ単位に調べる．

スライディングウィンドウ

符号化済みバッファと先読みバッファのシフトする文字数が異なるので，図 8に示すように，2 つの環状バッファとして構成する．符号化済みバッファのサイズは，符号の中に送信終了時の符号を入れることを考慮して $N-1$ とし，先読みバッファのサイズを Fとする．

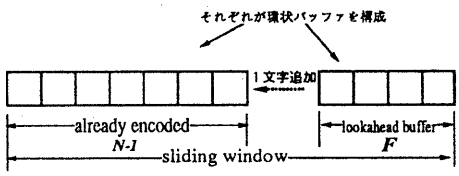

図 8 提案するアルゴリズムで使用するウィンドウ

```
while 先読みバッファが空でない do
  符号化済みバッファの中から先読みバッファと最も長く
    一致する文字列へのポインタ<offset, length>
    を求める
  if length > p then
    ポインタ<offset, length>を出力
    if <offset, length>は始めて符号化された then
      offset から length 文字を辞書に登録する
    endif
    符号化した先読みバッファの文字列を
    インデックスで置換
  else
    先読みバッファの先頭の 1 文字を出力
  endif
  符号化済みバッファへ 1 文字シフト
  先読みバッファは符号化した文字列の長さシフト
```

図 9 提案する符号化アルゴリズム

3.3 アルゴリズム

提案するアルゴリズムにおける符号は，一致した文字列へのオフセット i と符号化済みバッファにおける一致した文字列の長さ jの 2 つの組$< i, j >$とする．符号化済みバッファには辞書へのインデックスも含まれるので，LZSS における文字列の長さとこの jの値が同じであっても実質的に表される長さは異なる．

提案するアルゴリズムにおいても，LZSS と同様に一致する文字列の最大長を決める必要がある．LZSS では，Fであるが，ここでは，先読みバッファを可変長としているため，L と表す．LZSS において，一致した文字列の長さ jは，$p < j \leq F$となるのに対して，提案するアルゴリズムでは，$p < j \leq L$ となる．したがって，符号長は

$$L = 1 + \lceil \log_2 N \rceil + \lceil \log_2 (L - p) \rceil \qquad (2)$$

と表される．

図 9に，提案する圧縮アルゴリズムの符号化アルゴリズムを示す．復号化アルゴリズムも同様に構成できる．

たとえば，図 3 に示した文字列をさらに長くした文字列 “abcabcbacbabcabcbabcabcbc ···” を符号化した場合について図 10に示す．まず，“a”, “b”, “c” がそのまま出力され，“abc” は符号化済みバッファの 3 文字と一致するので，$< 3,3 >$と符号化され “abc” が辞書に登録される．その後 “b”, “a”, “c” と出力が続き，次の “ba” は$< 3,2 >$と符号化され辞書に登録される．さらに$< 1,2 >, < 7,2 >$と符号が続く．

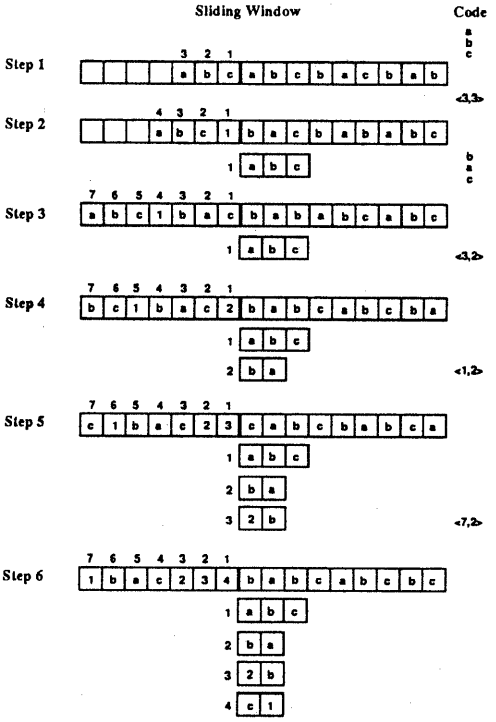

図 10 提案するアルゴリズムの符号化例

このように，フレーズの概念を導入し辞書を構成することにより，一致するパターンを増加させることがでる．さらに，先読みバッファの長さを増加させることにより，長いマッチング文字列を得ることができ，1 文字あたりの符号長を減少させることができる．

4 アルゴリズムの評価

提案するアルゴリズムの符号長は，符号化済みバッファのサイズ Nとマッチングする最長一致文字列長 L に依存するため，まず，その 2 つの値を変化させたときの圧縮率の変化を計測した．表 1には，サンプルデータとして UNIX(SunOS4.1.2) で使用されている 70861bytes の csh の日本語マニュアルを

表1 バッファサイズを変化させたときの提案するアルゴリズムの圧縮率

	$N=2^8$	$N=2^9$	$N=2^{10}$	$N=2^{11}$	$N=2^{12}$	$N=2^{13}$	$N=2^{14}$
$L=2^1$	40559 (57.2%)	36677 (51.8%)	32368 (45.7%)	29719 (41.9%)	27794 (39.2%)	27562 (38.9%)	28618 (40.4%)
$L=2^2$	38656 (54.6%)	34978 (49.4%)	30648 (43.3%)	28524 (40.3%)	27043 (38.2%)	26728 (37.7%)	27478 (38.8%)
$L=2^3$	39015 (55.1%)	35502 (50.1%)	31411 (44.3%)	29397 (41.5%)	27837 (39.3%)	27502 (38.8%)	28217 (39.8%)
$L=2^4$	40242 (56.8%)	36832 (52.0%)	32836 (46.3%)	30777 (43.4%)	29214 (41.2%)	28880 (40.8%)	29557 (41.7%)

用いた場合の計測結果を示しており，上段に圧縮後のバイト数，下段に圧縮率を示している．

ここで，圧縮率は式(3)により計算している．

$$[\text{圧縮率}] = \frac{\text{圧縮後のファイルサイズ}\,(bytes)}{\text{圧縮前のファイルサイズ}\,(bytes)} \times 100(\%) \quad (3)$$

この表より，$N=2^{13}=4096, L=2^2=4$ のとき最良の圧縮率を示すことが分かる．

また，種類の異なる他のテキストファイル（英語マニュアル，ソースプログラム，バイナリファイル）の圧縮を行ったときも上記の N と L の値での圧縮率が最良となった．

LZSS では圧縮率が最も良くなるときの最大一致文字列の長さである F の値は $F=18$ であることが知られているので，提案するアルゴリズムの最良の L の値 $(L=4)$ と比べてかなり大きな値となる．したがって，提案するアルゴリズムでは符号長を節約することが可能となる．

次に，LZSS アルゴリズムと提案するアルゴリズムの圧縮率を比較する．ここでは，最大一致文字列の長さの値は双方の最良な値の中間をとり，LZSS では $F=8$ とし，提案するアルゴリズムでは $L=8$ とする．$N=4096$ として日本語の csh のマニュアルを 10 等分し，先頭のブロックから順に圧縮するファイルを増加させたときの圧縮率の変化を図11に示す．ここで，LZSS による圧縮率の変化を実線で示し，提案するアルゴリズムの圧縮率の変化を点線で示している．

この図から明らかなように，提案するアルゴリズムによって約4%から10%まで圧縮率が向上し，ファイルサイズが大きくなるにつれてその効果が顕著になっている．

さらに，静的 Huffman 符号，LZW，LZSS における圧縮率と提案するアルゴリズムの圧縮率の比較を表2に示す．ここで，上段の値は圧縮後のファイルサイズを，下段の値は圧縮率を示している．これらの数値は各アルゴリズムが最良に動作する場合のパラメータを用いて計測している．

表2 各アルゴリズムにおける圧縮率の比較

アルゴリズム	サンプルデータ 実行型 155648	和文 70854	英文 76390	C ソース 45105
静的 Huffman 符号	111714 (71.7%)	46724 (65.9%)	41071 (53.7%)	29125 (64.6%)
LZW	123932 (79.6%)	64794 (91.4%)	29036 (38.0%)	22066 (48.9%)
LZSS	91068 (58.5%)	29521 (41.6%)	30673 (40.1%)	24047 (47.6%)
提案するアルゴリズム	84937 (54.5%)	26728 (37.7%)	28366 (37.1%)	23397 (46.3%)

この表からも明らかなように，各種のテキストファイルに対して提案するアルゴリズムは他のアルゴリズムより約4%圧縮率を向上させることができる．

5 おわりに

本論文では，LZSS 技法に基づく適応型データ圧縮アルゴリズムを提案した．

提案したアルゴリズムは LZW 技法におけるフレーズの概念を有効に採り入れ，LZSS より約4%高い圧縮率を実現することができた．

今後は LZSS における符号化済みバッファの冗長性などを特徴付け，提案するアルゴリズムの有効性を理論的に解析する必要がある．

参考文献

[1] J.Ziv and A. Lempel, "A Universal Algorithm

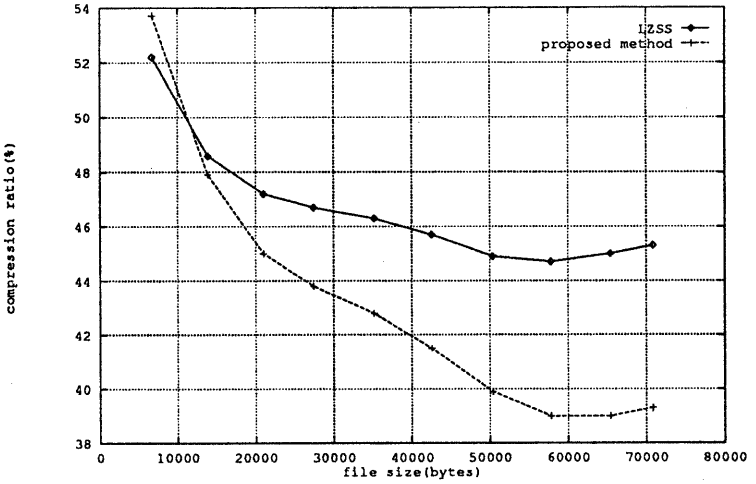

図 11 LZSS と提案するアルゴリズムの圧縮率の比較

for Sequential Data Compression" *IEEE Trans. Inf. Theory*,23,3(May), pp.337–343, 1977.

[2] J.Ziv and A. Lempel, "Compression of individual Sequence via Variable-rate Coding" *IEEE Trans. Inf. Theory*,24,5(September), pp.530–536,1978.

[3] J.A.Storer and T.G.Szymanski, "Data Compression via Textual Substitution" *J. ACM* 29, 4(Oct.),pp.928–951,1982.

[4] T.A.Welch, "A Technique for high-performance data compression" *Computer*17,6(June),pp.8–19,1984.

[5] UNIX, "UNIX User's Manual", Version 4.2. Berkeley Software Distribution, Virtual VAX-11 Version, Univ. of California, Berkeley, Calif, 1984.

[6] T.C. Bell, "Better OPM/L Text Compression", *IEEE Trans. Communications*, COM-34(12),pp.1176–1182,December,1986.

[7] T.C. Bell, J.G. Cleary, L.H. Witten "Text Compression", Prentice Hall, New Jersey, 1990.

[8] Debra A. Lelewer , Daniel S. Hirschberg, "Data Compression", *ACM Computing Surveys*,Vol.19, No.3, pp.261–296,1987