ユニバーサルデータ圧縮アルゴリズムとその改良方法について
Universal Data Compression Algorithms and their Improvements

山本博資
Hirosuke YAMAMOTO

電気通信大学 電子情報学科
Dept. of Communications and Systems
University of Electro-Communications

Abstract　情報源の確率パラメータに依存しない符号化復号化アルゴリズムを用いたデータ圧縮符号は, ユニバーサル符号と呼ばれており, Lempel–Ziv 符号を始めとして数多くのユニバーサル符号が提案され, 既にその幾つかは実用化されている. 本稿ではそのようなユニバーサル符号について解説するが, 各ユニバーサル符号のアルゴリズムを個別に紹介するのではなく, 全てのユニバーサル符号を統一的に取り扱い, その共通の概念を明らかにすると共に, 高性能なユニバーサル符号を構成するための一般的なテクニックを具体例を交えながら解説する

1　はじめに

通信や記録を効率よく行うためにデータ圧縮技術が用いられているが, 最近**ユニバーサル符号**が特に注目され, いろいろなところで実用化され始めている. あるクラスに属する全ての情報源出力に対して漸近的に最適な圧縮率を達成するユニバーサル符号の研究は古くから行われていたが [58][59]etc., それらは実用的な観点というよりは理論的興味から研究がなされていた. ユニバーサル符号が盛に研究される切っ掛けを作ったのは, Ziv と Lempel がいわゆる **Lempel-Ziv 符号** (あるいは Ziv-Lempel) 符号 [60]-[62] を提案したことによるが, かれらの符号もまた符号化定理を証明するために考案されたものであり, 効率の良いものではなかった. その後, Welch[66] が Lempel-Ziv 符号を改良したいわゆる LZW 符号を提案し, それが高速でかつ効率のよい圧縮方法であることを示した. またそれを少し改良した圧縮方式が UNIX の compress コマンド [67] としてインプリメントされたことにより, 実用的な符号としてユニバーサルデータ圧縮符号が広く知られるようになった.

その後, より効率がよく, より高速な圧縮を実現するために, 多くのユニバーサルデータ圧縮アルゴリズムや改良アルゴリズムが提案され, また既に多くの解説書 (解説論文)[1]-[9] も出版されている. しかし, それらの解説書は個々のアルゴリズムの紹介に留まっているものが多く, ユニバーサル符号全体を統一的に解説したものは少ない.

本稿では, 個別のアルゴリズムを紹介することよりも, ユニバーサル符号全体に共通する基本的な概念および効率を良くする共通的なテクニックを紹介することに主眼をおいて解説する. また, それら

を実現する方法として個別のアルゴリズムがどのように用いられているかを紹介する.

なお, 本稿では Lempel-Ziv 符号を始めとする **Dictionary 法** (あるいは Textual Substitution 法, Macro 法) と呼ばれているユニバーサル符号を取り扱う. ユニバーサル符号には Dictionary 法以外に, **Dynamic Huffman 符号** [16]-[24] や**算術符号 [10]etc. と Context-Modeling** (あるいは, MDL[13] などを用いた **Universal-Modeling**) を用いて構成する方法などがあるが, 本稿では取り扱わない. しかし, Dictionary 法の各符号と等価な文脈モデルを作ることにより, 「算術符号+文脈モデル」で Dictionary 法の符号と等価な符号を作れることに注意する必要がある [6][11]-[15].

2 性能評価方法

ユニバーサルデータ圧縮符号は, 当然よい圧縮率が達成できるものほど性能がよいことになるが, 各種のユニバーサル符号の性能を公正に比較するためには次のような項目に関して性能を比較しなければならない.

(a-1) 圧縮率

(a-2) 速度 (符号化速度, 復号化速度)

(a-3) 必要なメモリ量 (符号化メモリ量, 復号化メモリ量)

(a-4) そのユニバーサル符号が適用可能な情報源クラスの広さ

これらの項目に関して全てに優れているユニバーサル符号は存在せず, 一方の項目の性能を上げると他方の性能が悪くなるという trade-off の関係が一般に成り立っている.

どの項目の性能を良くすべきかは, その使用目的により異なる. たとえば, 「性能の低い端末から高性能計算機へのデータ通信」では「符号化速度, 符号化メモリ量の性能」が, また逆に「高性能計算機から性能の低い端末へのデータ通信」では「復号化速度, 復号化メモリ量の性能」が重要になる. また, 計算機のディスク上に記録された頻繁に使用するファイルを圧縮するような場合は, その圧縮されたファイルを瞬時に復号する必要があるが, 符号化は計算機が使用されていない時 (あるいは負荷の低い時) に時間をかけて圧縮することができる. 従って, そのような場合は復号時間の短い符号が必要となる.

(a-4) に関しては, より広いクラスを対象にすれば一般に圧縮率が悪くなり, 狭いクラスを対象にすればそのクラスに特有な性質を利用して圧縮率のよい符号を作りやすくなる. 例えば, 日本語の文書ファイルに対象を制限すれば, JIS コードと日本語の性質を利用して日本語文書に対して効率よく圧縮できるアルゴリズムを作れる [25]. しかし, ある特殊なクラスに対して効率をあげるように工夫すると, 一般的に言って, そのクラスを外れた情報源に対して圧縮率がかなり悪くなったり, 場合によっては正しく復号できなかったりする.

3 Dictionary 法

Dictionary 法によるユニバーサル符号では, 辞書を用いてデータ系列を 2 値系列に圧縮するわけであるが, 次のような 2 段階の手順で 2 値系列に符号化される.

(Step-A1) データ系列を正整数値[1] 系列に符号化する.
つまり, データ系列を部分系列に分解し, その各部分系列が辞書のどの部分に一致しているかを示す整数値を用いて, その部分系列を符号化する.

(Step-A2) 正整数値系列を 2 値系列に符号化する.
つまり, [Step-A1] で得られた各正整数を 2 値符号化する.

従って, よい圧縮率を達成するためには, 上記の各段階においてそれぞれ効率のよいアルゴリズムを用いなければならないが, 当然互いに関連しており, 整合性のよいアルゴリズムを用いる必要がある.

[1] または非負整数値

上記 (Step-A2) の整数を 2 値系列に符号化する方法には, 主に次の 3 通りの方法がある.

(b-1) 算術符号を用いる方法

(b-2) Dynamic Huffman 符号を用いる方法

(b-3) 正整数のユニバーサル表現を用いる方法

一般的に言って, (b-1)→(b-3) の順序で圧縮率は悪くなるが, 逆に処理速度は速くなり必要なメモリ量も少なくてすむ. したがってこれらのうちのどの方法を用いるかは, 圧縮率, 処理速度, 必要メモリ量の trade-off の関係により決めることになる. 次節ではこれらのうち, (b-3) の正整数のユニバーサル表現について少し詳しく説明する.

4 正整数のユニバーサル表現

Dictionary 法では, 前節で述べたようにまずデータ系列を正整数値系列に符号化する. この時, 各正整数は次のどちらかの性質を満たすように工夫されている場合が多い.

(c-1) 整数 n がある既知の N に対して, $0 \le n \le N-1$ の範囲で等確率で生起する.[2]

(c-2) 小さい正整数 n ほど大きい確率で生起する.

(c-1) の場合, 最も簡単な符号化法は $\lceil \log_2 N \rceil$ ビットの**固定長符号**で n を 2 進数表現する方法である. しかし, この方法では, 理想的な符号長 $\log_2 N$ に比べて, $(\lceil \log_2 N \rceil - \log_2 N)$ ビットのロスが生じる. このロスの最大の原因は, 固定長符号を符号木で表した時, 完全木にならないためであるが, この欠点を改良して完全木になるように改良した **CBT 符号** (Complete Binary Tree Code)[69][70][76][82] が知られている. CBT 符号化法の例を次に示す.

(Step-B1) $k = \lceil \log_2 N \rceil$ とする.

(Step-B2)
- $0 \le n \le 2^k - N - 1$ の時, n を $k-1$ ビットの 2 進数で表す.
- $2^k - N \le n \le N-1$ の時, $n + 2^k - N$ を k ビットの 2 進数で表す.

復号は次のように行える. 最初の $k-1$ ビット読み込み, それが $2^k - N - 1$ 以下なら, その $k-1$ ビットが整数 n を表しており, $2^k - N - 1$ より大きければ, もう 1 ビット読み込んだ合計 k ビットの数値から $2^k - N$ 引けば n が求まる.

データ圧縮の符号化が進むにつれて N が 1 つずつ増加していくような場合には, 上記の改良による圧縮率の改善効果は, あらく見積もって $(0.5/k) \times 100$ パーセント程度である. 従って, 符号化が開始した直後などで, N の値が小さい場合には改善効果が大きく, また N が数千程度 (k が 12 程度) に大きくなった場合でも, 数パーセント程度の改善効果がある. よって, 処理速度が問題にならない場合は, 固定長符号より CBT 符号を用いるべきである.

次に (c-2) の場合を考える. これは

$$P(n) \ge P(n+1),\ n \ge 1 \qquad (1)$$

を満たすような正整数 n を効率よく 2 値符号化する場合であり,

(d-1) $1 \le n \le N$ (予め n の最大値 N が分かっている場合)

(d-2) $1 \le n$ (予め最大値が分からない場合や, 分かっていても最大値が非常に大きい場合)

の 2 通りの場合に分けて考えるとができる.[3]

(d-1) の方式としては幾つかの方式が考えられるが [56], 簡単で効率のよい方式として **(start,step,stop) 符号** [82] がある. これは, ℓ ビットの 2 進数 (2^ℓ 個の数字を表すことができる) を $\ell = start$ から始めて, $\ell = start + step$, $\ell = start + 2 \times step$, $\cdots$, $\ell = stop$ まで, 順に用いていく方法である. ℓ ビットの 2 進数だけを書いたのでは, prefix 条件を満たさず, 符号語の区切りが分からなくなる. そこで ℓ ビットの 2 進数の前に $s = (\ell - start)/step$ を示す値を Unary 符号で示す. 例えば, (1,2,7) 符号の場合は, $\ell = 1, 3, 5, 7$

[2] n が $1 \le n \le N$ の場合は $n' = n - 1$ に置きかえて考える.

[3] n が $0 \le n$ の場合は $n' = n + 1$ に置きかえて考える.

のビット数の 2 進数が順に用いられ, 各 ℓ に対応した $s = 0, 1, 2, 3$ の値表す Unary 符号語 1,01,001,000 がその 2 進数の前に付けられて 2 値符号となる[4] . (1,2,7) 符号の例を表 1 に示す. なお, (2,1,10) 符号は Run-length 符号の Wyle-Erb-Banow 符号 [57] にほぼ等価となる.

表 1: Start-Step-Stop Code for (1,2,7)

Integer	code
0	1 0
1	1 1
2	01 000
3	01 001
4	01 010
⋮	⋮
9	01 111
10	001 00000
11	001 00001
12	001 00010
⋮	⋮
41	001 11111
42	000 0000000
43	000 0000001
44	000 0000010
⋮	⋮
169	000 1111111

(d-2) の場合に対する 2 値符号化法は, Prefix 条件を満たす (符号語の区切りを示す) 方法により, 大きく次の 2 つに分割できる [43][44][5] .

(e-1) 区切りを示す特殊な Flag を Suffix として付加する方式 [34]-[44]

(e-2) データ部 (数値を表している部分) のビット長を示す符号を Prefix としてデータ部の前に付加する方式 [26]-[33][47]-[55]

これらの方式の多くは漸近的 (確率 $P(n)$ によって定まるエントロピー $H(P(n))$ が十分大きい場合) に最適であることが示されている. しかし, 実際のユニバーサル符号で現れるエントロピー $H(P(n))$ はそれほど大きくはなく, 実用に向いていないものも多い. 以下では, 実用的に効率のよい 3 つの 2 値符号化を紹介する.

Capocelli の Fibonacci 方式 [42]

$F_\ell^{(f)}$ を次式で定義される f 次の Fibonacci 数とする.

$$F_i^{(f)} = \begin{cases} 0, & \text{if } i \le 0 \\ 1, & \text{if } i = 1 \\ F_{i-1}^{(f)} + F_{i-2}^{(f)} + \ldots + F_{i-f}^{(f)}, & \text{if } i \ge 2. \end{cases} \tag{2}$$

また, Fibonacci 数の和を次式で定義する.

$$S_i^{(f)} = \sum_{\ell=1}^{i} F_\ell^{(f)}, \quad i \ge 1 \tag{3}$$

$$M_i^{(f)} = \sum_{\ell=1}^{i} S_\ell^{(f)}, \quad i \ge 1 \tag{4}$$

この時, $n(\ge 1)$ を次のように符号化する.

$$B_{Fib(f)}(n) = \begin{cases} 01^{f-1}, & \text{if } n = 1 \\ 0B_{Fib(f)}(n - S_{i-1}^{(f-1)}), & \text{if } M_{i-1}^{(f-1)} < n \le M_{i-1}^{(f-1)} + F_i^{(f-1)} \\ 1B_{Fib(f)}(n - S_i^{(f-1)}), & \text{if } M_{i-1}^{(f-1)} + F_i^{(f-1)} < n \le M_i^{(f-1)} \end{cases} \tag{5}$$

ここで, 1^{f-1} は長さ $f-1$ の 1 の連を示す. 符号語の例を表 2 に示す. 表から分かるように, 01^{f-1} が符号語の区切りを示す flag として働いており, n が大きくなるにつれて, Prefix 部には 01^{f-1} を含まない全てのパターンが短い順に現れる. f は flag 長を定めるパラメータになるが, 実際のユニバーサル符号に用いる場合, $f = 3$(または 4) の時に効率がよくなることが多い. なお, $B_{Fib(f)}(n) = a_m a_{m-1} \cdots a_2 a_1 01^{f-1}$ から n は次式によって復号できる.

$$n = 1 + \sum_{i=1}^{m} S_{i+a_i}^{(f-1)} \tag{6}$$

Yamamoto-Ochi の Flag 方式 [44]

符号語の区切りを示す Flag として 10^{f-1} を用いて, $n(\ge 1)$ を次のように符号化する.

[4] $S = 3$ の時, 0001 とすべきであるが, 最大が 3 であるため最後の 1 は不要.

[5] これらの性能比較は, 文献 [33][44] を参照せよ.

(Step-C1) n の通常の 2 進数表示から, 常に 1 である MSB を除いたものを $[n]_-$ とする.

(Step-C2) $[n]_-$ を左から調べ, 10^{f-2} のパターンが存在したらその後ろに 1 を挿入する.

(Step-C3) 最後に flag パターン 10^{f-1} を付加する.

(Step-C2) で, flag パターンに一致する可能性が生じた時に, 1 を挿入することにより, flag パターンがデータ部に現れないようにしている. この方式の符号語 $B_{Flag(f)}(n)$ を表 2 に示す. この方式でも flag 長 f は自由に選ぶことができるが, 実用上は f=3(または 4) で効率が最も良くなる場合が多い. なお復号は次のように行えばよい. 10^{f-1} が現れた時, 次のビットが 1 なら, その 1 は挿入されたビットなので取り除く. また, 次のビットが 0 ならその 0 を合わせた 10^f は区切り記号である.

表 2 より, $P(n) \geq P(n+1)$ を完全に満たす分布に対しては, Fiboncacci 方式の方が Flag 方式より効率がよい. しかし, その効率の差はわずかであり, Flag 方式では Bit-stuffing(unstuffing) で符号化 (復号化) が行えるため, Fibonacci 方式に比べて, 高速に処理することができる.

Amemiya-Yamamoto の方式 [33]

上記の 2 つの方式は (e-1) に分類される方式であるが, これらとほぼ同じ性能を持つ (e-2) の符号化方式として次の符号化方式がある.

(Step-D1) n の通常の 2 進数表示から, 常に 1 である MSB を除いたものを $[n]_-$ とする.

(Step-D2) $[n]_-$ の桁数を $p(=\lfloor \log_2 n \rfloor)$ とする.

(Step-D3) p を 2^k で割った余りを $r(=p \bmod 2^k)$ とし, k 桁を用いた r の 2 進数表示を $(r)_k$ とする. また, その時の商を $q(=\lfloor \frac{p}{2^k} \rfloor)$ とする.

(Step-D4) 符号語を $B_{CE(k)}(n) = 1^q 0 (r)_k [n]_-$ とする.

ここで, k はパラメータである. 復号は $1^q 0 (r)_k$ から $[n]_-$ のビット長 p を求めることにより容易に行える. 表 2 に符号語の例を示す. $k = f - 1$ で Capocelli の Fibonacci 方式や Yamamoto-Ochi 方式とほぼ等しい性能となる. また, Amemiya-Yammaoto 方式は, 整数 n に対して $B_{CE(k)}(n)$ が符号語長順序, 大小関係, 辞書式順序を保存した符号となる. つまり, n に対して符号語長は単調非減少であり, $B_{CE(k)}(n)$ を普通の 2 進数とみなして大小関係を比較しても, $B_{CE(k)}(n)$ を先頭ビットから辞書式順序 $(0 < 1)$ で比較しても, n と同じ大小関係を満たしている.

表 2: Codewords $B_{Fib(f)}(n)$, $B_{Flag(f)}(n)$, $B_{CE(k)}(n)$

n	$B_{Fib(3)}(n)$	$B_{Flag(3)}(n)$	$B_{CE(2)}(n)$
1	011	100	0 00
2	0 011	0 100	0 01 0
3	1 011	1 100	0 01 1
4	00 011	00 100	0 10 00
5	01 011	010 100	0 10 01
6	10 011	10 100	0 10 10
7	11 011	11 100	0 10 11
8	000 011	000 100	0 11 000
9	001 011	0010 100	0 11 001
10	010 011	0100 100	0 11 010
11	100 011	0101 100	0 11 011
12	101 011	100 100	0 11 100
13	110 011	1010 100	0 11 101
14	111 011	110 100	0 11 110
15	0000 011	111 100	0 11 111
16	0001 011	0000 100	10 00 0000
17	0010 011	00010 100	10 00 0001
18	0100 011	00100 100	10 00 0010
19	0101 011	00101 100	10 00 0011
20	1000 011	01000 100	10 00 0100

5 辞書の構成法と符号化法

第 3 節の (c-1) に示したように, データ系列は辞書を用いて整数値に符号化されるが, ユニバーサルな圧縮が行えるようにデータに基づいて辞書をアダプティブに更新していく必要がある. 従って

(Step-E1) まだ符号化が終わっていないデータの次の部分系列を, 現時点での辞書で符号化する.

(Step-E2) (Step-E1) で符号化した部分系列を利用して, 辞書を更新する.

を繰り返しながら圧縮することになる. 復号側も同様に

(Step-F1) 現時点の辞書を利用して, 次の符号語から次の部分系列を復号する.

(Step-F2) (Step-F1) で復号化した部分系列を利用して, 辞書を更新する.

を繰り返しながら復号する. その結果, 辞書を送らなくても, 符号化側と同じ辞書が復号側にも作成でき, 正しく復号できることになる.

辞書を構成する時, ユニバーサル符号が満たさなければならない性能仕様 (a-1)-(a-4) に基づいて

(g-1) 辞書のデータ構造

(g-2) 辞書の参照方法 (符号化方法)

(g-3) 辞書の初期値, 辞書への単語 (データの部分系列) の登録方法, 単語の削除方法

などを決定しなければならないが, 第 2 節で述べたように (a-1)-(a-4) には trade-off の関係が存在するため, 唯一の最適な方式が存在する訳ではない. また, 言うまでもなく (g-1)-(g-3) は互いに関連しており, それぞれを別々に取り扱えるものではないが, 以下では, (g-1)-(g-3) に関してそれぞれ一般的な考え方をまとめておく.

辞書のデータ構造

辞書のデータ構造としては, Unsorted List, Sorted List, Link Buffer, Binary Tree, Trie, Hash table, etc. や, それらを組み合せたものが考えられる. どのデータ構造を用いるかは, (a-1)-(a-4) のどの性能に重きを置くかにより異なるが, 一般的に言って, 符号化を高速化するために, Tree や Trie 構造がとられることが多い. その例は, 第 6 節節で紹介する.

辞書の参照方法

参照できる単語 (データの部分系列) の長さにより,

(h-1) 固定長単語方式 [77][81][84]etc.

(h-2) 可変長単語方式

に分類することができるが, 実用的なユニバーサル符号の大半は可変長単語方式となっている. また, 辞書内の単語の参照方法により, 次の 2 通りの方式に分類することができる.

(i-1) 単語を辞書内の場所とその長さで指定する方式

(i-2) 単語の場所を指定するとその長さが一意に定まる方式

(h-1) の固定長単語方式は当然 (i-2) 方式となるが, (h-2) の可変長方式でも, 単語長の増加がある一定の規則にしたがって変化している場合は, (i-2) の方式が可能となる [62][66]etc.

辞書が, 既に符号化が済んだ部分系列を蓄えている単純な Buffer の場合に対して, (i-1) と (i-2) の符号化方式の例を示す. 簡単のために部分系列を $x(s,t) = x_s x_{s+1} \cdots x_t$ および $x(s,\cdots) = x_s x_{s+1} \cdots$ で表すことにする. 今, $x(1,m-1)$ の符号化が既に終了していて Buffer に蓄えられており, 次に $x(m,\cdots)$ を符号化する場合を考える. (i-1)(i-2) に対応して, それぞれ次のような符号化法を考えることができる.

(例 1) $x(m,\cdots)$ に最も長く一致する部分系列を Buffer 内で探した結果, ある s, $1 \le s \le m-1$, に対して, $x(m, m+\ell-1) = x(s, s+\ell-1)$ である時, x_m と x_s のインターバル値 $m-s$ と, 一致した長さ ℓ を用いて $\{m-s, \ell\}$ と符号化する [65]etc.

(例 2) $x(1, m-1)$ をある決った規則で, $x(1, m-1) = x(1, t_1)x(t_1+1, t_2)\cdots x(t_{u-1}+1, t_u)$ と分解しておく. この時, ある j, $j \le u$, に対して, $x(m, m+\ell-1) = x(t_{j-1}+1, t_j)$, $\ell = t_j - t_{j-1}$, の時, $x(m, m+\ell-1)$ を $\{j\}$ と符号化する.

$x(1,m)$ を分解する方法はいろいろ考えることができるが, 例えば,

$$x(t_{j-1}+1, t_j) = x(t_{i-1}+1, t_i)y, \text{ for some } i, 0 \le i \le j-1 \tag{7}$$

$$x(t_{j-1}+1,t_j) \neq x(t_i+1,t_{i+1}),\ \text{for any } i, 0 \leq i \leq j-1 \tag{8}$$

を満たすように分割する方法がある. ただし. $x(t_{-1}+1,t_0) = \lambda$(空系列) であり, y は $x(t_{j-1}+1,t_j)$ の最後のシンボルを表している. この分解法は, Lempel-Ziv[62] により提案され, **増分分解 (Incremental Parsing)** とよばれている.

(i-1) の方式は, 一つの部分系列を辞書に登録すれば, その長さを変えて参照できるため, 辞書の利用効率がよい. しかし, 場所と長さの 2 つのパラメータを用いて符号化しなければならないので, 符号化効率 (圧縮率) が悪くなる場合もある. 他方, (i-2) の方式は場所のパラメータだけで, 辞書内の単語を参照できるため, 符号化効率はよいが, 辞書に単語がまだ十分登録されていない場合 (符号化の初期段階など) で効率が悪くなる場合が多い.

辞書の初期値

辞書の初期値に関しては,

(j-1) 空 (または, ほぼ空の状態) の辞書

(j-2) 予め用意した比較的大きな辞書

から始める二通りが考えられる. 現在提案されているユニバーサル符号はユニバーサル性を高めるために, (j-1) の方式が取られているものがほとんどである. 本稿でも, (j-1) の方式のみ取り扱うことにするが, 今後 CD-ROM などを利用した標準の辞書が各計算機で利用できるような環境が整った場合, 後者の方式も検討する必要があろう.

単語 (部分系列) の登録方法

(Step-E2) で述べたように, 部分系列 $x(s,t)$ の符号化が終了すると, $x(s,t)$ に基づいて辞書に新たな単語を登録する必要がある. 単語の登録方法は, 辞書のデータ構造や辞書の参照方法に大きく依存しているが, ほぼ次のように分類できる.

(k-1) $x(s,t)$ に対して, $x(i,i+\ell-1)$, $s \leq i \leq t-1$, $1 \leq \ell$ (または,$1 \leq \ell \leq L$)[6] を全て登録する [61]etc.

(k-2) $x(s,t)$ (あるいは $x(s,t)y$)[7] のみを単語として登録する [62][66]etc.

(k-3) (k-1) と (k-2) の中間の方式. いろいろな登録法が可能であるが, 例えば, 次のような登録法が知られている.

(k-3-1) $x(s,t)$ に対して, $x(s,\cdots)$ (つまり, $x(s,s+\ell-1)$, $\ell = 1,2,\cdots$) を単語として登録する [82]etc.

(k-3-2) 各 k ごとに, $x(k,k+\ell_k) = x(m,m+\ell_m)y$, $0 \leq m < k$ (ただし, $x(0,0) = \lambda$(空系列), $\ell_0 = 0$) を最も長い ℓ_k で満たす部分系列を単語に登録する [93].

(k-1)(k-3-1) は (i-1) に対応した方式であり, (k-2)(k-3-2) は (i-2) に対応した方式である. また, (例 1)(例 2) はそれぞれ (k-1)(k-2) に対応している.

(i-2)(k-2) を用いた方式では, 登録された各単語がほぼ等確率で参照されると見なせる場合が多いので, (c-1) の 2 値符号化法を利用するものが多い [62][66]etc. 一方, (k-1) あるいは (k-3-1) に (i-1) を用いた方式では, 辞書内の場所と長さを用いて符号化することになるが, 一致する単語の長さは短い方が長い場合より生起しやすいので, (c-2) の 2 値符号化法が用いられる [60][74]etc. 場所を表す数字は (k-1) では (c-2) が用いられるが, (k-3) のように登録する部分系列を制限した場合は, Tree 構造を用いて (c-1) の 2 値符号化法に適した辞書にすることもできる [82][93].

単語の削除方法

通常符号化が進むにつれて, 単語が次々に登録されていくため, 辞書 (メモリ容量) が一杯になった時の処理を考える必要がある. 一般に, 辞書が一杯になった時の処理法としては次のような方法がある [80].

(l-1) 辞書を Reset する.

(l-2) 辞書を固定する. (つまり, 単語の登録を中止する.)

[6] ある与えられた最大値長 L 以下の全ての部分系列 [60][82]etc.

[7] y は $x(s,t)$ の次のシンボル

(l-3) Swap用の辞書を用意する. これは, 辞書が一杯になった時点 (あるいは辞書の容量がある基準値を超えた時点) で新しい辞書を作り始め, その辞書がある程度充実した時点で辞書をSwapする方式である.

(l-4) 辞書の一部を削除しながら新しい単語を登録する. この方式はどの単語を削除するかによって, さらに次のように分類できる.

- **(l-4-1)** 最も以前に登録された単語を消去する [74]etc.
- **(l-4-2)** 最も以前に参照された (LRU, Least Recently Used) 単語を消去する [76]etc.
- **(l-4-3)** 最も参照された回数の少ない (LFU, Least Frequently Used) 単語を消去する [79]etc.

これらの方式にはそれぞれ次のような欠点がある. (l-1) の方式は, Reset直後の圧縮率が悪くなる. (l-2) では, 辞書の固定後は適応性がなくなるため, 固定後にデータの特性が変化すると圧縮率が悪くなる. (l-3) では辞書を二重に用意するため, メモリの半分 (あるいは何割りか) を有効に利用できない. (l-4) は適応性を保ったまま, メモリを有効に利用できるので, 最もすぐれた方式ではあるが, 辞書がWindow-Bufferからできている簡単な場合などを除いて, 一般に処理が複雑になり, 符号化復号化時間が長くかかる欠点がある. コンピュータのメモリ量は年々大容量化してきているので, メモリ容量が十分大きくとれ, ファイルサイズが極端に大きくない場合は, (l-2) の固定方式をとれば十分な性能が得られる場合が多い.

6 木構造を用いた辞書の例

符号化操作では, 符号化したい部分系列と一致する系列を辞書内から検索する必要がある. (h-1) の固定長単語方式では, 全ての可能なパターンに対応した配列を用意し, 各パターンの出現時刻を記録していけば, 何時刻前に一致した部分系列が出たかを容易に調べることができる [81]. しかし, (h-2) の可変長単語方式では, できるだけ長く一致する部分系列を高速に見つける必要があるため, 辞書としてTree構造が用いられる場合が多い. 本節では, そのうちの (k-1) と (k-3-1) の場合にそれぞれ対応したBellとFiala-Greeneの構成法を紹介する.

二分探索木を用いた辞書の例 (Bell[74])

Bellは, LZSS符号化アルゴリズム [65](基本的には (例1) の符号化法) を二分探索木を用いて高速に符号化する方法を示している.

(例1) のように, $x(1, m-1)$ が既に符号化が終了しており, $x(m, \cdots)$ を符号化する場合を考える. ただし, 最大一致長を L 以下に制限した場合を考える. つまり, $x(m, m+L-1)$ にできるだけ長く一致するものを $x(1, m+L-2)$ 内で探すことを考える. これは, $x(m, m+L-1)$ と $x(s, s+L-1)$ を, $1 \le s \le m-1$ の全てぜ s に対して比較し, 最も長く一致するものを求めることで実現できる. この $1 \le s \le m-1$ までの $m-1$ 回の比較を高速に行うために, $x(s, s+L-1)$, $1 \le s \le m-1$, を辞書式順序で二分木上に並べたものを考える.

例として, $x(1, \cdots) = bcbacbababc \cdots$ で $L = 4$ の時に, $x(8, 11) = babc$ と最も長く一致する系列を探す場合を考える.

$$
\begin{array}{lll}
x(1,4) = bcba & x(2,5) = cbac & x(3,6) = bacb \\
x(4,7) = acba & x(5,8) = cbab & x(6,9) = baba \\
x(7,10) = abab &&
\end{array}
$$

が図1のように二分木上に辞書式順序を満たすように記憶されているとする. この時, $x(8, 11)$ と最も長く一致する系列の探索と, $x(8, 11)$ の二分木への登録とが次のように同時に行える. $x(8, 11)$ を二分木上を根から辞書式順序に基づいて比較しながら辿り, 挿入すべき場所を決める. この時 $x(8, 11)$ に対する最大一致系列は, 根から新たに挿入された $x(8, 11)$ へ辿るパス上に存在する次のどちらかの系列となる.

- $x(8, 11)$ が挿入された直前の節点の系列 ($x(6, 9) = baba$).

- パスの方向が最後に変化した節点の直前の節点の系列 ($x(3,6) = bacb$)

従って, この 2 つの系列のうち, 長く一致している方を探せば, 最大一致系列が求まる.

実際のデータ構造としては, Buffer サイズが有限となるように Link Buffer(先頭と最後が繋がった Buffer) にデータ系列を保存し, 二分木の各節点はその Link Buffer に対する Pointer を記憶しておけばよい. 符号化が進むと古いデータは Link Buffer から消えて行くが, それに対応して二分木上の古い節点を辞書式順序を乱さないように消去する必要がある.

二分探索を高速にするためには, 二分木の深さが一様になっている方がよい. そのために, 少し工夫すれば, 二分木が常に AVL 木 (平衡木) となるようにすることもできる. しかし, 極端に偏ったデータ系列でない限り, AVL にする工夫を行わなくても, 通常はほぼつりあった二分木が自然に構成されていく.

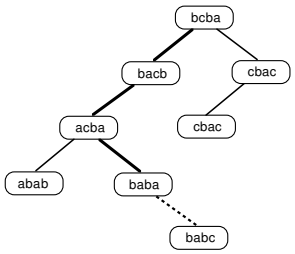

図 1: Bell's binary tree

二分探索木を用いた辞書の例 (Fiala-Greene[82])

Fiala-Greene 符号には A,B,C の 3 種類の符号が存在し, A は (k-1) に B,C は (k-3-1) に対応した符号になっている. ここでは, 符号化が簡単な B,C 方式について紹介する. Fiala-Greene 符号は, データ系列を Patricia Tree と Link Buffer を用いて記憶する. Patricia Tree では各枝に長さ 1 以上の文字列を対応させており, その対応を内部節点と葉で次のように記憶する.

- 内部節点には, 根からその節点までのパスに対応した部分系列の Link Buffer 上での開始位置とその長さを記憶する.
- 葉には, 根からその葉までのパスに対応した部分系列の Link Buffer 上での開始位置を記憶する.

葉に長さを記憶しないのは, 葉に至るパスは無限に長い系列に対応していると考えるためである.

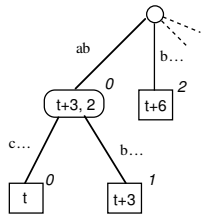

図 2: Patricia tree

例として, $x(t, t+9) = abcabbbacc$ に対応して図 2 の Patricia Tree ができているものとする. (簡単のため, $x(1, t-1)$ に対応した部分は図の省略した所に対応しているものとする.) 根から各葉までのパスが $x(t, \cdots)$, $x(t+3, \cdots)$, $x(t+6, \cdots)$ に対応している. また, 根から内部節点までのパスは $x(t, t+1) = x(t+3, t+4) = ab$ に対応しているが, 番号の新しい $x(t+3, t+4)$ に対応したデータ (t+3 から始まって長さ 2) の (t+3,2) を内部節点に記憶する. 図の状態で, $x(t+10, \cdots) = abbbc\cdots$ を符号化する場合, Patricia Tree を abbbc の順に辿ると, 左から 2 番目の葉に繋がっている枝の途中まで一致していることがわかる. その葉に記憶されている番号 t+3 と根からの一致していた文字数 4 から, $x(t+10, t+14)$ が $x(t+3, t+6)$ に一致していることがわかる.

Fiala-Greene の B 符号では, $x(t+10, t+14)$ が $x(t+3, t+6)$ に一致していることを示すのに, Link Buffer 上の一致の開始位置 t+3 と一致系列長 4 を用いて, {t+3,4} と符号化する. しかし, $x(t{+}10, t{+}14)$ は Patricia tree 上で左から 2 番目の葉に繋がっている枝の 2 番目の文字まで一致していると表すこともできる. Fiala-Greene 符号の C 符号では, 各葉と各節点にそれぞれ別の通し番号を振っておき, 例えば左から 2 番目の葉の番号が *1* なら, その番号 *1* と分岐した節点からの一致長 2 を用いて, $x(t{+}10, t{+}14)$ を {*1*,2} と符号化する. Patricia Tree 上の節点や葉の番号を利用すると, Link Buffer 上の番号を利用

するより小さい正整数で符号化でき, また一致長も, 最後に分岐した節点からの一致長だけを記述すればよいので, 小さい正整数で表すことができる. このことは, それらの正整数を (c-2) を用いて 2 値系列に符号化する時, 短いビット数ですむことを意味しており, 圧縮効率がよくなる.(第 7 節参照)

$x(t{+}10, t{+}14)$ の符号化が済むと, $x(t{+}10, t{+}14)$ を Patricia Tree に登録することになるが, $x(t{+}3, t{+}6)$ の部分で枝分かれさせて, 図 3 のようになる. この時, 各節点に記憶してある情報を最も新しい一致番号に変更する. また, Buffer として Link Buffer を用いているので, Buffer から消えてしまった部分系列を Patricia Tree から消去する作業も必要となる.

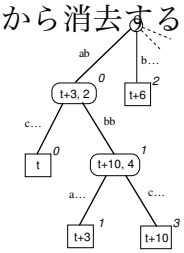

図 3: Patricia tree

7 圧縮率の改良

各種のユニバーサル符号に関して, 圧縮率を改善する数多くのアイデアが出されているが, 本稿では多くのユニバーサル符号に共通に適用できる圧縮率の改善方法について幾つか紹介する.

第 5 節の (例 1) の符号化法では, データの部分系列 $x(m, m{+}\ell{-}1)$ が, $x(m, m{+}\ell{-}1) = x(s, s{+}\ell{-}1)$, $s < m$, を満たす時, $x(m, m{+}\ell{-}1)$ をそのインターバル長 $m - s$ と一致系列長 ℓ を用いて, $\{m - s, \ell\}$ と符号化する. 通常, これらの整数値は小さい方が生起しやすいので, (c-2) に対応した 2 値符号化法を用いて 2 値系列に符号化される. したがって, そのような場合は, インターバル長と一致系列長をそれぞれ $m - s, \ell$ より小さい正整数で表現できれば, より効率よく圧縮できることになる. 以下では, インターバル長と一致系列長のより小さい正整数を用いた表現方法について紹介する.

一致系列長の短縮化表現

インターバル長 $m - s$ が分かっている時, 一致系列長 ℓ をより小さい数字で表現する方法を考える. $x(s, s + \ell - 1)$ は, $x(m, m + \ell - 1)$ に最も近い一致系列なので, $x(t, t + \ell - 1) \neq x(m, m + \ell - 1)$, $s < t < m$, であるが, $x(t, t + \ell' - 1) = x(m, m + \ell' - 1)$, $s < t < m$, $0 \leq \ell' < \ell$, を満たす t と ℓ' が存在する. このような ℓ' の中で最も長いものを ℓ_M とすると, $\{m - s, \ell\}$ の代わりに, $\{m - s, \ell - \ell_M\}$ と表現しても, $x(m, m + \ell - 1) = x(s, s + \ell - 1)$ を復元できる [88].

例えば, $x(1, 14) = cabcdaabcaabcd$ の場合, $x(11, 14)$ が $x(2, 5)$ に等しいので, $x(11, 14)$ をインターバル長と一致系列長を用いて符号化すると $\{9, 4\}$ となるが, $x(11, 13) = x(7, 9) = abc$ より, $\ell - \ell_M = 4 - 3 = 1$ であることを利用すると $\{9, 1\}$ と符号化できる. この時, $\{9, 1\}$ より実際の一致系列長は次のように求まる. $\{9, 1\}$ と $m = 11$ から $x(s, s + \ell - 1)$ が $s = 2$ から始まる系列であることがわかる. 次に x_2 から始まる系列と最も長く一致する系列 $x(t, t + \ell_M - 1)$ を $3 \leq t \leq m - 1$ の範囲で探す. この例の場合は, $x(2, 4) = x(7, 9) = abc$ が最大一致系列となり, $\ell_M = 3$ であることが分かる. したがって, $\ell - \ell_M = 1$ より, $\ell = 4$ であることが分かる.

この符号化方は, 第 6 節で紹介した Fiala-Greene 符号の C 方式で取り入れられいる. C 方式では, 一致系列の長さを Patricia Tree 上で最後に枝分かれした節点からの長さで表しているが, これは $\ell - \ell_M$ を表しているのに他ならない.

インターバル長の短縮化表現 1

次に一致長 ℓ が分かっている時に, インターバル長を小さい正整数で表現する方法について考える.

$x(m, m + \ell - 1)$ に一致するものを探すときに, s が $s < m - \ell + 1$ の範囲の $x(s, s + \ell - 1)$ のみに制限をする (つまり, $x(m, m + \ell - 1)$ と重ならない範囲だけに制限する) と, $\{m - s - \ell + 1, \ell\}$[8] の代わりに, $x(s, s + \ell - 1)$ が $x(m, m + \ell - 1)$ から, 長さ ℓ の系列として何種類前の系列かを示す Recency-Rank(RR

[8] $m - s$ から $\ell - 1$ を引いているのは, $s < m - \ell + 1$ の範囲に制限しているため

と略す) 値を用いて, $\{RR, \ell\}$ と符号化できる [77].

例えば, $x(1,15) = abcdaaaaaaabcd$ で $x(12,15)$ を符号化する場合, $x(12,15) = x(2,5) = abcd$, $s = 2$, $m = 12$, $\ell = 4$ より, $x(12,15)$ のインターバル長 $m-s-\ell+1$ を用いた符号化では $\{7,4\}$ と符号化される. これに対して, $x(2,5)$ と $x(12,15)$ の間には, $bcda$, $cdaa$, $daaa$, $aaaa$ の 4 種類の長さ $\ell = 4$ の部分系列が存在し, $x(2,5)$ は $x(12,15)$ に対して 5 種類目 ($RR = 5$) の系列になるので, RR 値を用いた表現では $\{5,4\}$ となる.

RR 値は常にインターバル値以下になるので, RR 値を用いればインターバル値より小さい正整数で表現できる. RR 値を用いる方法は特に ℓ が小さい場合に効果がある.

ℓ が固定長の場合, RR 値はワープロなどで用いられている move-to-front 方式 (出現した単語を辞書の一番前に並び換える方式) を用いて求めることができる [63][73]. つまり, 全ての種類の単語を一列に並べた逐次リストを用意し, 単語 $\mathbf{x}^\ell$ が出現するごとに, その単語 $\mathbf{x}^\ell$ を move-to-front 方式で逐次リストの先頭に移動する. この時, 単語 $\mathbf{x}^\ell$ の RR 値は, その逐次リストにおける単語 $\mathbf{x}^\ell$ の順位で求まる.

逐次リストを用いる方法としては, move-to-front 方式以外に, 出現した単語 $\mathbf{x}^\ell$ を逐次リスト上で k 個前に移動する move-ahead-k 方式や, その k が $k = 1$ の場合に相当する transpose 方式などを用いることも可能である [79][83].

インターバル長の短縮化表現 2

$x(1, m-1)$ を利用して, $x(m, m+\ell-1)$ を符号化する時, (例 1) では x_m から $x_{m+\ell-1}$ までの系列の繋がりやすさを考慮した符号化になっているが, $x(1, m-1)$ と $x(m, m+\ell-1)$ にまたがる系列の繋がりやすさを全く考慮していない. これを考慮に入れた符号化法として, 次のような方式を考えることができる [87][92].

$x(m, m+\ell-1)$ を符号化する時, $x_{m-1}x(m, m+\ell-1)$ と一致する系列 $x_{s-1}x(s, s+\ell-1)$ を探す. $x_{m-1} = x$ の時, そのインターバルを文字 x に続くところだけで数えあげる.

例えば $x(1,15) = abcdaabcacabcd$ に対して, $x(12,15) = abcd$ を符号化する場合を考える. (例 1) の符号化法では通常のインターバル値が 10 となり, $\{10,4\}$ と符号化される. これに対して, $x_{11} = c$ で条件を付けて探すと, c は x_9,x_4,x_1 で現れており, 3 個前の $c = x_1$ に繋がる $x_1x(2,5)$ が $x_{11}x(12,15)$ と一致している. したがって, $x_{11} = c$ で条件付けて数えたインターバル値 3 を用いて, $\{3,4\}$ と符号化できる.

上の説明では, x_{m-1} の一文字で条件付けることを考えたが, 一般に $x(m-j, m-1)$ の j 文字で条件付けることも可能である. 条件の長さ j を長くすると, その条件が出現する個数が少なくなるため, インターバルをより小さい数字で表現できる. しかし, $x(m-j, m-1)x(m, m+\ell-1)$ に一致する $x(s-j, s-1)x(s, s+\ell-1)$ を探すので, 条件の長さ j が長くなると一致系列長 ℓ の長い一致が存在する確率が小さくなり, 反って圧縮率が悪くなる.

上記の短縮化表現を実際の符号化法に組み込むには, 次のようにすればよい. 辞書を文字の種類分の複数個用意し, 一つ前のシンボル x_{m-1} がどのシンボルであるかに基づいて, 使う辞書を決めて符号化する [92].

LZW 符号や Fiala-Green 符号に適用した場合, 約 10K バイト前後以上の大きさのファイルでは, $j = 1$ で条件付けした (256 通りの辞書を用いた) 方が, 条件を付けない場合 (つまり単一の辞書を用いる場合) より効率がよくなる [98].

非圧縮モードの導入

部分系列 $x(m,n)$ が 2 値系列に圧縮符号化され時のビット系列を $B(x(m,n))$ で表す時,

$$|B(x(m,n))| < |x(m,n)| \tag{9}$$

を満たせば, $x(m,n)$ が圧縮符号化されたことになる. ただし, $|B(x(m,n))|$ は $B(x(m,n))$ のビット長であり, $|x(m,n)|$ は $x(m,n)$ を ASCII コードや JIS コードなどで表した時のビット長である. しかし, 場合によっては符号化を行うと式 (9) が満たされず, 反って長くなってしまう場合が起りえる. 特に, 符号化の初期などで辞書が十分成長していない場合などによく生じる.

このような膨脹を防ぐために, 通常非圧縮モードを導入し, 次のような符号化出力 $\tilde{B}(x(m,n))$ を出す [74][82]etc.

$$\tilde{B}(x(m,n)) = \begin{cases} 0B(x(m,n)), & \text{if (9) holds.} \\ 1x(m,n), & \text{otherwise} \end{cases} \tag{10}$$

$\tilde{B}(x(m,n))$ の 1 ビット目は, 圧縮モードであるか非圧縮モードあるかを区別するためのフラグである. 非圧縮モードを導入すると $B(x(m,n))$ だけを用いた場合に比べて, フラグに 1 ビット必要な分だけ効率が悪くなる. しかし, 上記のような膨脹を防ぐことができ, 全データ $x(1,\cdots)$ に対する圧縮率は, 非圧縮モードを用いない場合より非圧縮モードを導入した方が, 効率がよくなる場合が多い.

また, 辞書の更新に (l-4) のような単語の削除を許す場合は, 辞書内に一致する系列が全く存在しなくなる可能性がある. そのような場合には, 非圧縮モードは不可欠となる.

8 おわりに

本稿では, 各種のユニバーサルデータ圧縮符号に共通に存在する概念を明らかにし, 高性能な符号作るために必要な基本的な手法を幾つかの例を用いて紹介した. 紙面の都合上, 各ユニバーサル符号の個々のアルゴリズムには触れなかったが, 興味のある人は, 例えば文献 [2, Ch.8] などを参照して欲しい.

また, 本稿では取り上げなかったが, 重要なテーマとして, 非常に大きなアルファベットサイズのデータに対する圧縮法や, データ圧縮のハード化の問題などがある. 前者は, 16 ビットや 32 ビット程度の桁数で表した実数値データなどを符号化する時, アルファベットサイズが非常に大きいため, 新らたな部分系列が過去の部分系列と完全に一致することが少なく, 通常のユニバーサル符号のアルゴリズムでは効率よく圧縮できない. しかし, 実数値の上位桁だけを取り出せば, 同じ値が繰り返し出現する. このような特徴を利用すれば, アルファベットサイズが非常に大きなデータに対して効率よく圧縮する符号を作ることができる [85][91]. 後者に関しては, 多数のマイクロプロセッサで並列処理できるようなアルゴリズムを考案したり [72][97], 汎用のマイクロプロセッサの処理に適したアルゴリズムを考える必要がある [66].

本稿で示した (a)〜(l) の各項目を組み合せれば, 多種多様のユニバーサル符号が構成できる. しかし, 商業用のユニバーサルデータ圧縮符号を作る場合はともかく, ユニバーサル符号を学問的に研究する場合は, 圧縮率の細かい数字を競うよりも, 研究対象の符号がどのような点にどのような新しいアイデアが含まれているかを明らかにすることの方が意味がある. そのような意味付けを行う時に, 本稿で示したような分類が役に立つものと思われる.

参考文献

[1] J. Storer, *Data Compression*, Computer Science Press, 1988.

[2] T. Bell, J. G. Cleary and I.H.Witten, *Text Compression*, Prentice Hall, Inc, 1989.

[3] R. N. Williams, *Adaptive Data Compression*, Kluwer Academic Publishers, 1991.

[4] D. A. Lelewer and D. S. Hirschberg, "Data Compression" *ACM Computing Survey*, vol.19, no.3, pp.261–296, Sep. 1987.

[5] 山本博資, "計算機科学におけるデータ圧縮", **情報理論とその応用学会『データ圧縮』ワークショップ講演資料集**, pp.22–31, Jan. 1988.

[6] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for Text Compression" *ACM Computing Surveys*, vol.21, no.4, pp.557–591, Dec. 1989.

[7] 山本博資, "ユニバーサルデータ圧縮アルゴリズムについて", 信学会技報, vol.IT90, no.97, pp.7–14, Jan. 1991.

[8] 奥村, 吉崎, "圧縮アルゴリズム入門" *C Magazine*, vol.3, no.1, pp.44–68, Jan. 1991.

[9] 植松, 宮下, "プログラムやテキスト・データのための可逆圧縮アルゴリズムを理解する" インターフェース, pp.88-123, Aug. 1992.

[10] I .H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic Coding for Data Compression", *Communications of the ACM*, vol.30, no.6, pp.520–540, June 1987.

[11] J. Rissanen, “A Universal Data Compression Scheme”, *IEEE Trans. Inform. Thoery*, vol.IT-29, no.5, pp.656–664, Sep. 1983.

[12] G. .G. Langdon, Jr., “A Note on the Ziv-Lempel Model for Compressing Individual Sequences”, *IEEE Trans. Inform. Theory*, vol.IT-29, no.2, pp.284–287, March 1983.

[13] J. Rissanen, “Universal Coding, Information, Prediction, and Estimation”, *IEEE Trans. Inform. Theory*, vol.IT-30, no.4, pp.629–636, July 1984.

[14] 横尾英俊, “MDL 規準による適応的データ圧縮法のモデリング”, 信学会論文誌, vol.J69-A, no.8, pp.974–982, Aug. 1986.

[15] 朴, 今井, 山森, 伊藤, 石井, “パター ンマッチングと算術符号を用いた実用的なデータ圧縮法”, 信学会論文誌, vol.J71-A, no.8, pp.1615–1623, Aug. 1988.

[16] N. Faller, “An Adaptive System for Data Compression”, *Record of the 7-th Asilomar Conference on Circuits, Systems and Computers*, pp.593–597, 1973.

[17] R. G. Gallager, “Variations on a Theme by Huffman”, *IEEE Trans. Inform. Thoery*, vol.IT-24, no.6, pp.668–675, Nov. 1978.

[18] C. L. McMaster, “Documentation of the Compact Command”, *Unix User’s Manual*, 4.2BSD, VIrtual VAX-11 Version, Univ. of California, Berkeley, 1984

[19] D. E. Knuth, “Dynamic Huffman Coding”, *J. Algorithms*, vol.6, pp.163–180, 1985.

[20] J. S. Vitter, “Design and Analysis of Dynamic Huffman Coding”, *J. ACM*, vol.34, no.4, pp.825–845, Oct. 1987.

[21] J. S. Vitter, “ALGORITHM 673, Dynamic Huffman Coding”, *ACM Trans. Math. Softw.*, vol.15, no.2, pp.158–167, June 1989.

[22] H. Yokoo, “An Improvement of Dynamic Huffman Coding with a Simple Repetition Finder”, *IEEE Trans. Commun.*, vol.39, no.1, pp.8–10, Jan. 1991.

[23] G. V. Cormack and R. N. Horspool, “Algorithms for Adaptive Huffman Codes”, *Information Processing Letters*, vol.18, pp.159–165, March 1984.

[24] D. W. Jones, “Application of Splay Trees to Data Compression”, *Comm. of the ACM*, vol.31, no.8, pp.996-1007, Aug. 1988.

[25] 萩原剛志, “仮名文字の短縮表現を用いた日本語文書の圧縮について”, 信学会論文誌, vol.J74-A, no.9, pp.1431–1438, Sep. 1991.

[26] P. Elias, “Universal Codeword Sets and Representations of the Integers”, *IEEE Trans. Inform. Theory*, vol.IT–21, no.2, pp.194–203, March 1975.

[27] J. L. Bently and A. C. Yao, “An Almost Optimal Algorithm for Unbounded Search”, *Inform. Processing Letters*, vol.5, no.3, pp.82–87, Aug. 1976.

[28] S. Even and M. Rodeh, “Economical Encoding Commas between Strings”, *Commun. ACM*, vol.21, no.4, pp.315–317, April 1978.

[29] B. Y. Ryabko, “Encoding of a Source with Unknown but Ordered Probabilities”, *Prob. Inform. Transm.* 1979.

[30] D. E. Knuth, “Supernatural Numbers”, in D. A. Klarner (Ed.), *The Mathematical Gardner*, pp.310–325, Prindle Weber and Schmidt, Boston, 1980.

[31] Q. F. Shout, “Improved Prefix Encodings of the Natural Numbers”, *IEEE Trans. Inform. Theory*, vol.IT-26, no.5, pp.607–609, Sep. 1980.

[32] H. Yokoo, “An Efficient Representation of the Integers for the Distribution of Partial Quotients over the Continued Fractions”, *J. of Inform. Processing*, vol.11, no.4, pp.288–293, 1988.

[33] T. Amemiya and H. Yamamoto, “A New Class of the Universal Representation for the Positive Integers” *IEICE Trans.* (to appear).

[34] C. E. Shannon, “The Mathematical Theory of Communication”, *Bell Syst. Techn. J.*, vol.27, no.3, pp.379-423, July 1948.

[35] V. E. Hoggat and M. Bicknell, “Generalized Fibonacci Polynomials and Zeckendorf’s Representations”, *The Fibonacci Quarterly*, vol.11, no.3, pp.399-419, 1973.

[36] L. J. Guibas and A. M. Odlyzko, “Maximum Prefix-synchronized Code”, *SIAM J. Appl. Math.*, vol.35, no.2, pp.401–408, Sep. 1978.

[37] R. G. Stone, “On Encoding of Commas between Strings”, *Commun. ACM*, vol.22, no.5, pp.310–311, May 1979.

[38] K. B. Lakshmanan, “On Universal Codeword Sets”, *IEEE Trans. Inform. Theory*, vol.IT–27, no.5, pp.659–662, Sep. 1981.

[39] R. M. Capocelli, and A. De Santis, “Regular Universal Codeword Sets”, *IEEE Trans. Inform. Theory*, vol.IT–32, no.1, pp.129–133, Jan. 1986.

[40] A. Apostolico and A. S. Fraenkel, “Robust Transmission of Unbounded Strings Using Fibonacci Representations”, *IEEE Trans. Inform. Theory*, vol.IT–33, no.2, pp.238–245, March 1987.

[41] M. Wang, “Almost Asymptotically Optimal Flag Encoding of the Integers”, *IEEE Trans. Inform. Theory*, vol.IT–34, no.2, pp.324–326, March 1988.

[42] R. M. Capocelli, “Comments and Additions to ‘Robust Transmission of Unbounded Strings Using Fibonacci Representations’ ”, *IEEE Trans. Inform. Theory*, vol.35, no.1, pp.191–193, Jan. 1989.

[43] R. M. Capocelli, “Flag Encoding Related to the Zeckendorf Representation of Integers”, in R. M. Capocelli (Ed.) *Sequences*, *Combinatorics, Compression, Security, and Transmission*, pp.449-466, Springer-Verlag, 1990.

[44] H. Yamamoto and H. Ochi, “A New Asymptotically Optimal Code of the Positive Integers”, *IEEE Trans. Inform. Theory*, vol.37, no.5, pp.1420–1429, Sep. 1991

[45] Y. Miyakawa, Y.Saitoh, H.Imai, “New Binary Encoding Schemes of Positive Integers”, *IEICE Trans.*, vol.E74, no.9, pp.2504–2512, Sep. 1991.

[46] 中條, 稲積, 西島, 平澤, “自然数表現の一般化について”, 信学会論文誌, vol.J75-A, no.1, pp.94–100, Jan. 1992.

[47] 松井, 伊理, “あふれのない浮動少数点表示”, 情報処理学会論文誌, vol.21, no.4, pp.306-313, July 1980.

[48] 浜田穂積, “二重指数分割に基づくデータ長独立実数表現法”, 情報処理学会論文誌, vol.22, no.6, pp.521-526, Nov. 1981.

[49] 浜田穂積, “二重指数分割に基づくデータ長独立実数表現法 II”, 情報処理学会論文誌, vol.24, no.2, pp.149-156, March 1983.

[50] 浜田穂積, “計算機内部における数の表現法”, 情報処理学会誌, vol.24, no.4, pp.353-357, April 1983.

[51] 中森, 土井, “3 重指数分割による数値表現方式について”, 信学会論文誌, vol.J71-A, no.7, pp.1468-1469, July 1988.

[52] 中森眞理雄 “変動多重指数分割による数値表現方式”, 信学会論文誌, vol.J72-A, no.6, pp.1009-1011, June 1989.

[53] 横尾英俊, “自然数の表現の立場から見た多重指数分割浮動少数点表示方式”, 情報処理学会論文誌, vol.30, no.6, pp.792-794, June 1989.

[54] 横尾英俊, “自然数の表現に基づく多重指数分割浮動少数点表示方式のクラス”, 信学会論文誌, vol.J72-A, no.12, pp.1998-2004, Dec. 1989.

[55] 中森, 萩原, 高田, “変動多重分割による自然数表現法”, 情報処理学会論文誌, vol.31, no.6, pp.939-942, June 1990.

[56] 越智, 山本, “有限正整数に対する 2 値符号化法について”, **第 *12* 回情報理論とその応用シンポジウム予稿集**, pp.559-564, Dec. 1989.

[57] H. Wyle, T. Erb, and R. Barow, “Reduced-Time Facsimile Transmission by Digital Coding”, *IRE Trans. on Comm. Systems*, vol.CS-9, no.3, Sep. 1961.

[58] J. Ziv, “Coding Source with Unknown Statistic-Part I: Probability of Encoding Error”, *IEEE Trans. on Inform. Theory*, vol.IT-18, no.3, pp.384–394, May 1972.

[59] L. D. Davisson, “Universal Noiseless Coding”, *IEEE Trans. on Infrom. Theory*, vol.IT-19, no.6, pp.783–795, Nov. 1973.

[60] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression”, *IEEE Trans. Inform. Theory*, vol.IT-23, no.3, pp.337-343, May 1977.

[61] J.Ziv, “Coding Theorems for Individual Sequences”, *IEEE Trans. Inform. Theory*, *IEEE Trans. Inform. Theory*, vol.IT-24, no.4, pp.405-412, July 1978.

[62] J. Ziv and A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding”, *IEEE Trans. Inform. Theory*, vol.IT-24, no. 5, pp.530-536, Sep. 1978. (U.S. patent 4,464,650)

[63] B. Y. Ryabko, “Data Compression by Means of a ‘Book Stack’ ”, *Prob. Infom. Transm.* vol. 16, no.4, 1980 (English Translation, 1981).

[64] M. Rodeh, V. R. Pratt and S. Even, “Liner Algorithm for Data Compression via String Matching”, *J. of ACM*, vol.28, no.1 pp.16-24, Jan. 1981.

[65] J. A. Storer and T. G. Szymanski, “Data Compression via Textual Substitution”, *J. ACM*, vol.29, no.4, pp.928-951, Oct. 1982.

[66] T. A. Welch, “A Technique for High-Performance Data Compression”, *IEEE Computer*, vol.17, no.6, pp.8-19, June 1984.

[67] S. W. Thomas, J. McKie, S. Davies, K. Turkowski, J. A. Woods and J. . Orost, “Compress (version 4.0) program and documentation”, available from joe@petsd.UUCP.

[68] V. S. Miller and M. N. Wegman, “Variations on a Theme by Ziv and Lempel”, in *Combinatorial algorithms on word*, edited by Apostolico and Z. Galil, pp.131-140, NATO ASI Series, vol.F12, Springer-Verlag, 1985, (or *IEEE ICC’88*, pp.390-394, 1988).

[69] 山本, 中田, “Ziv-Lempel 符号の改良とシミュレーションによる性能評価-[II]”, **信学会技報告**, CS85-135, pp.1-8, Jan. 1985.

[70] 横尾英俊, “ユニバ-サル情報源符号化のための修正 Ziv-Lempel 符号”, **信学会論文誌**, vol.J68-A, no.7, pp.644-671, July 1985.

[71] M. Jakobsson, “Compression of Character Strings by an Adaptive Dictionary”, *BIT*, vol.25, no.4, pp.593-603, 1985.

[72] M. E. Gonzalez Smith and J. A. Storer, “Parallel Algorithms for Data Compression”, *J. ACM*, vol.32, no.2, pp.344-373, April 1985.

[73] J. L. Bentley, D. D. Sleator, R. E. Tarjan and V. K. Wei, “A Locally Adaptive Compression Scheme”, *Commun. ACM*, vol.29, no.4, pp.320-330, April 1986.

[74] T. C. Bell, “Better OPM/L Text Compression’, *IEEE Trans. Communication*, vol.COM-34, no.12, pp.1176-1182, Dec. 1986.

[75] T. C. Bell, “A Unifying Theory and Improvements for Existing Approaches to Text Compression”, *Ph.D. Thesis*, Dep. of Computer Science, Univ. of Canterbury, New Zeland, 1987.

[76] P. Tischer, “A Modified Lempel-Ziv-Welch Data Compression Scheme”, *Australian Computer Science Communications*, vol.9, no.1 pp.262-272, 1987.

[77] P. Elias, “Interval and Recency Rank Source Coding: Two On-line Adaptive Variable-Length Schemes”, *IEEE Trans. Inform. Theory*, vol.IT-33, no.1 pp.3-10, Jan. 1987.

[78] R. P. Brent, “A Linear Algorithm for Data Compression”, *Australian Computer J.*, vol.19, no.2, pp.64-68, May 1987.

[79] R. N. Horspool, “A Locally Adaptive Data Compression Scheme”, *Commun. of the ACM*, vol.30, no.9, p.792, Sep. 1987.

[80] J. A. Storer, “Parallel Algorithms for On-Line Data Compression”, *IEEE ICC’88*, pp.385-389, 1988.

[81] F. M. J. Willems, “Universal Data Compression and Repetition Times”, *IEEE Trans. Inform. Theory*, vol.35, no.1, pp.54-58, Jan. 1989.

[82] E. R. Fiala and D. H. Greene, “Data Compression with Finite Windows”, Comm. of the ACM, vol.32, no.4, pp.490-505, April, 1989.

[83] 朴, 高島, 今井 “Self-Organizing Rule に基づく適応的データ圧縮法”, **信学会論文誌**, vol.J72-A, no.8, pp.1353-1359, Aug. 1989.

[84] A. Wyner and J. Ziv, “Some Asymptotic Properties of the Entropy of a Stationary Ergodic Data Source with Applications to Data Compression”, *IEEE Trans. on Inform. Theory*, vol.35, no.6, pp.1250–1257, Nov. 1989.

[85] H. Yokoo, “A Lossless Coding Algorithm for the Compression of Numerical Data”, *The Trans. of The IEICE*, vol. E73, no.5, pp.638-643, May, 1990.

[86] 村島, 中村, 廣田, “文字列登録を強化した LZW 法によるソースファイルの圧縮”, 信学会論文誌, vol.J73, no. 9, pp.1529-1533, Sep. 1990.

[87] N. Merhav, “Universal Coding with Minimum Probability of Codeword Length Overflow”, *IEEE Trans. on Inform. Theory*, vol.37, no.3, pp.556–563, May 1991.

[88] P. E. Bender and J. K. Wolf, “New Asymptotic Bounds and Improvements on the Lempel–Ziv Data Compression Algorithm”, *IEEE Trans. on Inform. Theory*, vol.37, no.3, pp.721–729, May 1991.

[89] A. D. Wyner and J. Ziv “Fixed Data Base Version of the Lempel–Ziv Data Compression Algorithm”, *IEEE Trans on Inform. Theory*, vol.37, no.3, pp.878–880, May 1991.

[90] T. Kawabata and H. Yamamoto, “A New Implementation of Ziv-Lempel Incremental Parsing Algorithm”, *IEEE Trans. on Inform. Theory*, vol.37, no.5, pp.1439–1440, Sep. 1991.

[91] S. Itoh and N. Goto, “An Adaptive Noiseless Coding for Sources with Big Alphabet Size”, *IEICE Trans.*, vol.E74, no.9, pp.2495–2503, Sep. 1991.

[92] E. Plotnik, M. .J. Weiberger, and J. Ziv, “Upper Bounds on the Probability of Sequences Emitted by Finite-State Sources and on the Redundancy of the Lempel–Ziv Algorithm”, *IEEE Trans. on Inform. Theory*, vol.38, no.1, pp.66–72, Jan. 1992.

[93] H. Yokoo, “Improved Variations Relating the Ziv–Lempel and Weltch-Type Algorithms for Sequential Data Compression”, *IEEE Trans. on Inform. Theoy*, vol.38, no.1, pp.73–81, Jan. 1992.

[94] 森田, 小林, “制約つき再生可能な文字列分解に基づく計算機ファイルのデータ圧縮” 情報処理学会論文誌, vol.33, no.2, pp.110–121, Feb. 1992.

[95] Y. Amit and M. I. Miller, “Large Deviations for the Asymptotics of Zive–Lempel Codes for 2-D Gibbs Fields”, *IEEE Trans. on Inform. Theory*, vol.38, no.4, pp.1271–1275, July 1992.

[96] A. Nobel and A. D. Wyner, “A Recurrence Theorem for Dependent Processes with Applications to Data Compression”, *IEEE Trans. on Infrom. Theory*, vol.38, no.5, pp.1561–1564, Sep. 1992.

[97] N. Ranganathan and S. Henriques, “High Speed VLSI Design for Lempel–Ziv Based Data Compression”, *IEEE Trans. on Circuits and Systems* (Submitted).

[98] 大峰恵, 電気通信大学電子情報学科山本研究室 Meeting 資料, Nov. 1992.