愛知工業大学研究報告
第３３号Ｂ　平成10年

動的ハフマン符号を同期させた動的辞書法による１パスデータ圧縮

A One-Pass Data Compression by Dynamic Dictionary Method Synchronized with Dynamic Huffman Codes

伊藤 雅 †
Masaru ITOH

Abstract　This paper proposes a new data compression algorithm. In the dynamic dictionary method, when building the dictionary from an original file, a large number of single characters appear. Each character is usually put into a file with fixed-length bits. The performance of compression ratio can be improved by using the dynamic Huffman codes which is assigned to a part of this output procedure. The features of our algorithm are that it is a one-pass data compression and that it is reversible coding. It is necessary that both dictionary tree and dynamic Huffman tree must be synchronously updated, when encoding a sequential of characters from input file to output one. The algorithm utilizes LRU (Least Recently Used) queue for deleting a dictionary item when the dictionary is full.

1.　はじめに

データ圧縮とは，ある長さの入力データ列をそれが含む冗長度を利用してより短い圧縮データ列に変換することである．入力データ列の長さと圧縮データ列の長さの比を圧縮率（compression ratio）といい，圧縮率をできるだけ良くすることがデータ圧縮の本質である．データ圧縮の利点は通信コストの削減や記憶容量の節約などが挙げられ，データが大規模になればなる程その必要性は顕著になる．

データ圧縮の分類 [1] には種々あり，ひとつは可逆圧縮と非可逆圧縮である．可逆圧縮は主として実行ファイルやテキストファイルなどの圧縮に使われる．一方，非可逆圧縮は主として画像や音声データで使われる．データ圧縮にはもうひとつ，静的・動的という分類もある．静的符号化とはメッセージから符号への写像が符号化中，常に一定であるものをいい，多くが２パス方式を採用している．他方，動的符号化とはメッセージから符号への写像が時間的特性を反映するような１パス圧縮をいう．動的符号化の中でも特に，入力データ列の統計的性質に依らず，その局所性を利用した符号をユニバーサル符号という．

ユニバーサル符号を生成する代表的な圧縮法にレンペル・ジブ（Lempel-Ziv）圧縮 [2] がある．これは文字列の置き換え（textual substitution）によるものと増分分解（incremental parsing）によるものの２種類がある．これら２つの方法はそのアルゴリズムのイメージからそれぞれスライド辞書法（sliding dictionary method） [3]，動的辞書法（dynamic dictionary method） [4] とよばれる場合がある [5]．スライド辞書法は入力文字列と一致する部分文字列を先に読み込ませた文字列バッファの中からスライドさせる形で探し出し，その位置と長さで置き換えて圧縮を実現する．動的辞書法は入力文字列から既に辞書に登録されている最長一致文字列を探索し，その辞書項目で置き換えて圧縮を実現する．このとき参照した最長一致文字列を前回一致文字列の末尾に付加することによって新たな文字列を辞書に登録する．

無記憶情報源に対する符号化のひとつにハフマン法（Huffman coding） [2] がある．これは文字を常に固定長で表現するのではなく，出現頻度の高い文字にはより短い符号語を割り当てるというものである．そのためハフマン木を用いて語頭条件（prefix property）を満足する可変長符号を生成する必要がある．事前にすべての文字の出現頻度情報を必要とする静的ハフマン法（static Huffman coding）と動的に頻度を見積もりながら符号化する動的ハフマン法（dynamic Huffman coding）がある．

提案法の基本的な考え方は，入力データ列の冗長性・局所性に対処可能な動的辞書法の一部に動的ハフマン符号を取り入れるというものである．文字列を逐次辞書に登録していく動的辞書法の増分分解過程では，必ず切りこぼしとなる単一文字が多数発生する．単一文字は通常固定長で出力されることが多い．この部分に動的ハフマン符号を適用すれば，圧縮率を改善できる．この場合，単一文字を処理する毎に辞書とハフマン木の両方を同期させて動的に更新する必要がある．提案法は動的辞書法を基本としているため，ユニバーサル性を保持しており可逆符号を生成する．

†愛知工業大学 経営工学科 （豊田市）

2. 動的辞書法と動的ハフマン法の処理過程とその欠点

動的辞書法では{(0,文字コード),(1,辞書項目)}という要素の組合せで圧縮を進めていくのが通例である．動的辞書法の一つであるLZT（Lempel-Ziv-Tischer）圧縮[6) 7)]の処理過程では辞書木（dictionary tree）とLRU待ち行列（Least Recently Used queue）を用いる．LRU待ち行列は辞書が一杯になった場合に，辞書木の中で最も使われていない辞書項目を削除し，その項目を再利用するために利用される．以下ではこのLZTに基づく圧縮法を動的辞書法とよぶことにする．

動的辞書法のアルゴリズムの概略を以下に記す．

動的辞書法の符号化アルゴリズム

Step1: 事前に辞書木に全文字種を登録し，LRU待ち行列を空とする．ファイルポインタをファイルの先頭に設定し，前回一致文字列もなしとする．

Step2: 2文字以上の一致文字列が既に辞書木に登録されている場合には，それを検索し，辞書項目を(1,辞書項目)として書き出す．2文字以上の一致文字列が辞書木にない場合には，(0,文字コード)を書き出す．

Step3: 前回一致文字列の末尾に今回一致文字列を付加して辞書木の更新を行う．同時にLRU待ち行列の更新も行う．

Step4: ファイル終端ならば終了．さもなければファイルポインタを今回一致文字列長だけ進め，前回一致文字列を今回一致文字列で更新する．そしてStep2に戻り符号化を続ける．

Step2の文字コードの書き出し方法は固定長である．辞書項目の書き出し方法について説明する．文字種をN，現在までに登録されている辞書項目数を$SIZE$，符号化すべき辞書項目をDとする．このとき，Dの符号語は$(D-N)$を$\ln SIZE$ビットで2進表現して与えることができる．

復号過程は符号化アルゴリズムのStep2とStep4を次のように置き換えるだけである．

Step2: 読み取りビットが1ならば，続けて辞書項目を読み込み，現在の辞書木で元の文字列を復元する．読み取りビットが0ならば，続けて文字コードを固定長で読み込み，元の文字を復元する．

Step4: 元ファイルサイズに達すれば終了．さもなければStep2に戻り復号化を続ける．

例として，16バイト（128ビット）からなる文字列 example1 = "acba␣cba␣cb␣acb∧Z" を用いて説明する．ここで，'∧Z'はファイル終端を表す文字である．

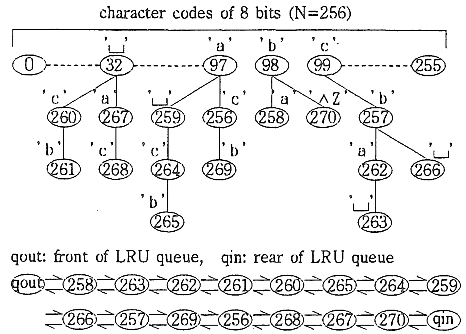

図1 example1の辞書木とLRU待ち行列

表1 辞書項目に対応する文字列

辞書項目	登録文字列
256	"ac"
257	"cb"
⋯	⋯
269	"acb"
270	"b∧Z"

動的辞書圧縮による結果は(0,'a') (0,'c) (0,'b') (0,'a') (0,'␣') (1,257) (1,259) (1,257) (0,'␣') (1,256) (0,'b') (0,'∧Z') となる．文字コードを8ビット，辞書項目を12ビットで表現するならば，example1は124ビットに圧縮されたことになる．圧縮終了時の辞書木とLRU待ち行列の最終状態を図1に，各辞書項目に対応する登録文字列を表1にそれぞれ示す．

動的辞書法の欠点はこの例のように単一文字'a', 'b', '␣'が複数回出現している点である．この部分を動的ハフマン符号で置換すれば，さらなる圧縮率の向上が期待できる．

一方の動的ハフマン法では{⟨path0,文字コード⟩,⟨path1⟩}という要素の組合せで圧縮が行われる．ここで，path0とはハフマン木の根から「未出現文字」ノードまでのパスであり，path1とは根から「登録済み文字」ノードまでのパスである.「未出現文字」とは現時点においてまだハフマン木に登録されていない未出現の文字すべての呼称である．以下では未出現文字ノードを0-nodeと表記する．

動的ハフマン法については，1985年に発表されたFGK（Faller-Gallager-Knuth）アルゴリズム[8)]と1989年に発表されたVitterアルゴリズム[9)]の2種類が有名である．どちらもPascal言語で書かれたソースコードが掲載されている．Vitterのハフマン木は木の高さを抑制できる点でFGKよりも優れている．しかし，そのために暗黙の番号（implicit number）と不変（invariant）条件という2つの概念をFGKに追加

する必要がある．動的辞書圧縮の一部に動的ハフマン符号が利用できることを示すのが本論文の目的である．よって以下では実装や拡張，さらには説明がより容易なFGKを基本とする．動的ハフマン法のアルゴリズムの概略を以下に記す．

動的ハフマン法の符号化アルゴリズム

Step1: ハフマン木として0-nodeのみを用意する．

Step2: 現在の符号化対象文字が未出現文字の場合には，根から0-nodeまでのパス⟨path0⟩をまず出力し，続いて符号化対象文字をCBT符号⟨文字コード⟩で出力する．他方，符号化対象文字が登録済み文字の場合には，根からその文字に対応する葉ノードまでのパス⟨path1⟩のみを出力する．

Step3: 符号化対象文字が未出現文字の場合には，その文字をハフマン木の葉ノードに追加登録する．対象文字が登録済み文字の場合には何もしない．

Step4: 対象文字の葉から根までの経路上にある各ノードの出現頻度をそれぞれ1ずつ増やしながらハフマン木を更新する．このとき兄弟条件（sibling property）[8]を常に満足しなければならない．

Step5: ファイル終端ならば終了．さもなければ次の文字を符号化対象文字とした後，Step2に戻る．

ここで，Step2の「CBT符号」とはComplete Binary Tree符号[7]の略であり，文献[8]と同一とした．すなわち，文字集合$\{a_1,\ldots,a_m\}$で$m=2^e+r$, $0\le r<2^e$のとき，$1\le k\le 2r$ならばa_kを$k-1$として$e+1$ビットで，$2r<k\le m$ならばa_kを$k-r-1$としてeビットでそれぞれ2進表現する．また，Step4の「兄弟条件」とはハフマン木の下から上，左から右の順序に各ノードの出現頻度が昇順に整列されていなければならない条件のことである．

復号過程は符号化アルゴリズムのStep2とStep5を次のように置き換えるだけである．

Step2: 読み取った0-1ビット列でハフマン木の0-nodeに到達する場合には，続けて文字コードをCBT符号で読み込み，元の文字を復元する．0-1ビット列で0-node以外の葉ノードに到達する場合には，その葉に登録されている文字を復元する．

Step5: 元ファイルサイズに達すれば終了．さもなければStep2に戻り復号化を続ける．

先のexample1に対する動的ハフマン圧縮による結果は⟨0,'a'⟩ ⟨0,'c'⟩ ⟨00,'b'⟩ ⟨0⟩ ⟨100,'␣'⟩ ⟨10⟩ ⟨111⟩ ⟨11⟩ ⟨001⟩ ⟨10⟩ ⟨01⟩ ⟨001⟩ ⟨11⟩ ⟨10⟩ ⟨01⟩ ⟨000,'∧Z'⟩ の74ビットとなる．圧縮終了時のハフマン木を図2に示す．

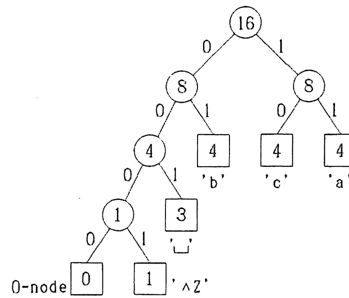

図2 example1のハフマン木

動的ハフマン法の最大の欠点は頻度情報しか利用しておらず，多くのファイルに見られる冗長性・局所性に全く対処できないことである．一方の動的辞書法はこの点を十分考慮しており，元ファイルサイズが無限大になれば，漸近的最適となることが知られている[1]．但し，example1のような小さな例ではこのような優位性は確認できない．

3. 動的ハフマン符号を同期させた動的辞書法による1パスデータ圧縮

2. のexample1を動的辞書法で圧縮したときの(0,文字コード)要素の文字コード部分だけを取り出すと，example2 = "acba␣␣b∧Z"の8バイト（64ビット）を得る．この部分に動的ハフマン法を適用すると，結果は⟨0,'a'⟩ ⟨0,'c'⟩ ⟨00,'b'⟩ ⟨0⟩ ⟨100,'␣'⟩ ⟨1101⟩ ⟨111⟩ ⟨000,'∧Z'⟩ となり，64ビットから58ビットに圧縮できる．

さて，example1を動的辞書圧縮した(0,文字コード)の部分にexample2の上記の符号を埋め込むと，結果は(0,⟨0,'a'⟩) (0,⟨0,'c'⟩) (0,⟨00,'b'⟩) (0,⟨0⟩) (0,⟨100,'␣'⟩) (1,257) (1,259) (1,257) (0,⟨1101⟩) (1,256) (0,⟨111⟩) (0,⟨000,'∧Z'⟩)となる．この結果，**2.** でexample1を動的辞書圧縮した124ビットがさらに118ビットに圧縮できる．この6ビットの差はexample2を動的ハフマン法で圧縮した場合の効果と全く同じである．

このことからファイルサイズが十分大きくなれば動的辞書法を単独で使用するよりも動的辞書法のルーチンに動的ハフマン符号を組み込んだ提案法の方が圧縮率の点で優れたものになることが予想される．

上記の例ではexample1からexample2を生成して最終的な符号化導出を試みている．しかし，このままでは動的辞書法がもつ1パス圧縮という利点が損なわれる．そこで，動的辞書圧縮の実行過程で動的ハフマン符号を同時に生成する機構が必要となる．本節の残りではこの観点から動的ハフマン符号を同期させた動的辞書法による1パスデータ圧縮について詳述する．

動的辞書法では辞書木とLRU待ち行列の2つを管

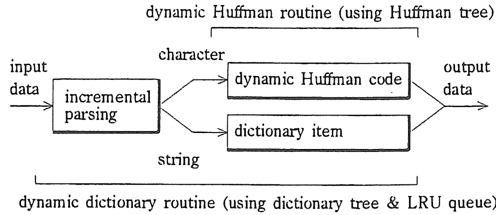

図3 提案法の処理過程概念図

理する必要がある．提案法では併せてハフマン木を動的辞書圧縮の処理過程で常に保持する必要がある．

具体的には動的辞書部で(0,文字コード)の要素をファイルに書き出す前に文字コードのみを動的ハフマン部に転送して現在のハフマン木から動的ハフマン符号を得て，この符号をファイルに書き出せばよいことになる．動的辞書部で(1,辞書項目)の要素を書き出す場合には動的ハフマン部を通さずそのまま書き出せばよい．図3は提案法の処理過程概念図である．アルゴリズムで使用する記号を整理しておく．

N: 文字種
i: ファイルポインタ
j: 処理回数
c_i: 第i番目の符号化対象文字
s_j: 最長一致文字列 ($s_j =$ "$c_i c_{i+1} \cdots c_{i+L_j-1}$")
L_j: 最長一致文字列長
s_{j-1}: 前回一致文字列
L_{j-1}: 前回一致文字列長
$D(s_j)$: 文字列s_jの辞書項目

提案法の符号化アルゴリズム

Step1: 初期化

事前に辞書木にN種の文字を登録し，LRU待ち行列を空とする．LRU待ち行列は双方向リストで表現し，先頭（出口）に$qout$，末尾（入口）にqinの番兵を用意する．前回一致文字列も空とし，ハフマン木は0-nodeのみとする．ファイルポインタiと処理回数jを$i=1, j=1$で初期化する．

Step2: 最長一致文字列の検索

現在の辞書木から最長一致文字列s_jを検索し，

if $L_j = 1$ and $c_i \in$ ハフマン木 $\Rightarrow$ $(0, \langle \text{path1} \rangle)$
if $L_j = 1$ and $c_i \notin$ ハフマン木 $\Rightarrow$ $(0, \langle \text{path0}, 'c_i' \rangle)$
if $L_j > 1 \Rightarrow (1, D(s_j))$

を場合に応じてそれぞれ書き出す．ここで，path0，path1とは**2.**の動的ハフマン法の符号化アルゴリズムで定義したものと同一である．

Step3: 辞書木の更新

$t_1 =$ "$s_{j-1} c_i$"
$t_2 =$ "$s_{j-1} c_i c_{i+1}$"
$\cdots$

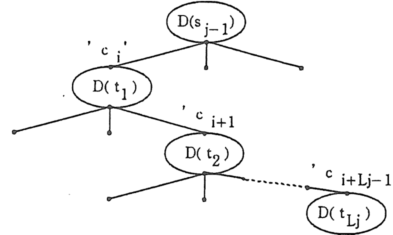

図4 辞書木内での辞書項目間の関係

$t_{L_j} =$ "$s_{j-1} c_i c_{i+1} \cdots c_{i+L_j-1}$"

というL_j個の文字列のうち辞書項目として未登録の文字列がある場合にはそれらを辞書木に追加登録する．辞書項目には図1にあるようにN以降を順次割り当てる．辞書木内での辞書項目間の関係を図4に示す．

Step4: LRU待ち行列の更新

$L_{j-1} = 1$の場合には最長一致文字列辞書項目$D(s_j)$をLRU待ち行列の末尾に移動させ，さらにL_j個の辞書項目を末尾に追加する．すなわち，

$$qout \leftleftharpoons \cdots \leftleftharpoons D(s_j) \leftleftharpoons \overbrace{D(t_{L_j}) \cdots D(t_1)}^{L_j\text{個の辞書項目}} \leftleftharpoons qin$$

一方，$L_{j-1} > 1$の場合には最長一致文字列辞書項目$D(s_j)$をLRU待ち行列の末尾に移動させ，L_j個の辞書項目を前回一致文字列辞書項目$D(s_{j-1})$の直前に追加挿入する．すなわち，

$$qout \leftleftharpoons \cdots \leftleftharpoons \overbrace{D(t_{L_j}) \cdots D(t_1)}^{L_j\text{個の辞書項目}} \leftleftharpoons D(s_{j-1}) \leftleftharpoons \cdots \leftleftharpoons D(s_j) \leftleftharpoons qin$$

Step5: ハフマン木の更新

上記Step2で(1,辞書項目)を出力した場合は何もしない．(0,動的ハフマン符号)を出力した場合のみハフマン木を更新する．更新は**2.**動的ハフマン法の符号化アルゴリズムStep3~4と同様である．

Step6: 終了判定

ファイル終端ならば終了．さもなければ$i = i + L_j$, $s_{j-1} = s_j$, $j = j+1$と更新後，Step2に戻る．

復号過程は符号化アルゴリズムのStep2とStep6を次のように置き換えるだけである．

Step2: 文字列または単一文字の復元

読み取りビットが1ならば，続けて辞書項目を読み込み，現在の辞書木で元の文字列を復元する．読み取りビットが0ならば，さらに0-1ビット列を読み込み，ハフマン木の0-nodeに到達する場合には，続けて文字コードをCBT符号で読み込み，単一文字を復元する．0-1ビット列が0-node

以外の葉に到達する場合には，その葉に登録されている単一文字を復元する．

Step6: 終了判定

元ファイルサイズに達すれば終了．さもなければStep2に戻り復号化を続ける．

辞書木が一杯になった場合の処理について記す．上述の符号化アルゴリズムに従えば，LRU待ち行列の先頭$qout$から末尾qinまでは順序よく辞書木の葉から根までの最も使われていない辞書項目が整列している．

辞書木の更新過程で辞書が一杯となりｍ個の辞書項目が不足する場合を考える．一杯になった時点のLRU待ち行列が以下のようになっていたとする．

$$qout \rightleftharpoons \overbrace{D(s_{j_1})\cdots D(s_{j_m})}^{m\text{ 個の辞書項目}} \rightleftharpoons \cdots \rightleftharpoons qin$$

このとき，$qout$からqinに向かってｍ個の辞書項目$D(s_{j_1})$ ～ $D(s_{j_m})$を辞書木とLRU待ち行列から順に削除する．これによって文字列s_{j_1} ～ s_{j_m}が今後参照不能となるが，新たにｍ個の文字列t_{j_1} ～ t_{j_m}が辞書木に登録可能となる．このようにすべての辞書項目をLRU待ち行列で管理し，辞書項目の削除と再利用を可能にする．

4. 実験結果

Microsoft C/C++ Ver.7.0で実装した提案法のプログラムサイズは約44Kバイトである．辞書サイズは4Kバイトとした．提案法（以下PROGと略記）の性能評価のため，動的辞書法（以下LZTと略記）と動的ハフマン法（以下FGKと略記）の両者で比較検討した．FGKのコードは文献[8]のPascalコードに基づいており，LZTは文献[5]に基づいている．いずれもC言語のポインタ版に修正したものを使用した．開発したPROGとのプログラム上の整合性をとり，対等の条件で正当に評価するためである．LZTとFGKを比較対象とした理由は次の二点による．第一に提案法は両者の融合を図っているということ．第二に提案法が１パス圧縮方式であるため，比較対象も１パス方式の圧縮法の方が妥当であると判断したためである．

性能評価はNEC PC-98（Cx486DLC，42MHz）上でさまざまなサイズのテキストファイル（94編）と実行ファイル（108編）の２群について実施した．それぞれのファイル群でのPROG，FGK，LZTによる圧縮率を図5～図8に示す．圧縮率は元ファイルと圧縮後ファイルのサイズの比とした．

LZTに対する提案法の優位性を検証するため，横軸xにテキストファイルサイズを，縦軸yに$(LZT - PROG)$すなわち圧縮後ファイルの出力バイト数の差をとって回帰分析を行った．結果を図9に示す．回帰係数は0.0122であった．同様の回帰分析を実行ファイルについても行った．また，FGKに対する提案法の

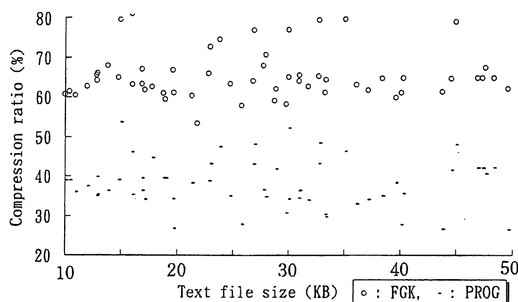

図５ テキストファイルでのPROGとFGKの圧縮率

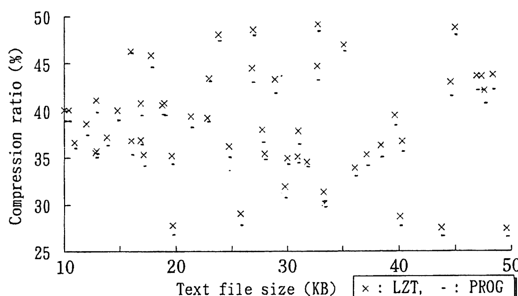

図６ テキストファイルでのPROGとLZTの圧縮率

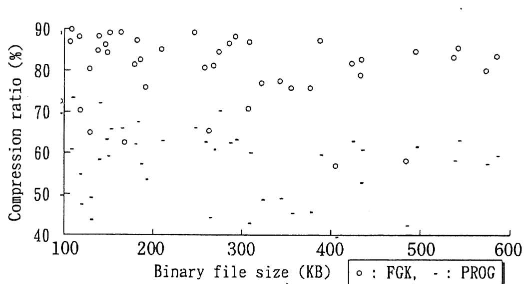

図７ 実行ファイルでのPROGとFGKの圧縮率

優位性も同様に検証した．提案法による圧縮率の改善を表2にまとめる．

これらの結果からテキストファイルに限定すれば，提案法はLZTと比較して圧縮率の面で1.22%程度の改善ができることを確認した．図9で3KB未満の比較的小規模なファイルについて，提案法の圧縮率がLZTよりも若干劣る場合がある．これは動的辞書圧縮の増分分解過程で切りこぼしとなる単一文字を動的ハフマン符号化する際にハフマン木情報を余分にファイルに書き出す必要があるからである．

次に実行時間について検討する．実行時間はRAMディスク上で各ファイル毎に計測した．ファイル群での比較が容易となるよう50KBに正規化した場合の圧縮および復元時間を表3に示す．表から復元時よりも

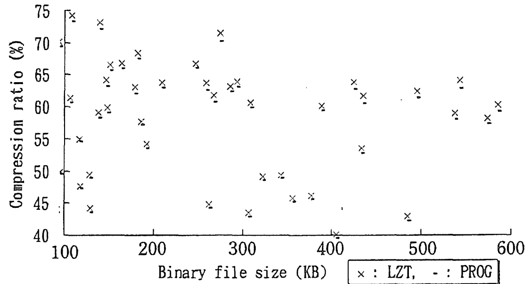

図8 実行ファイルでのPROGとLZTの圧縮率

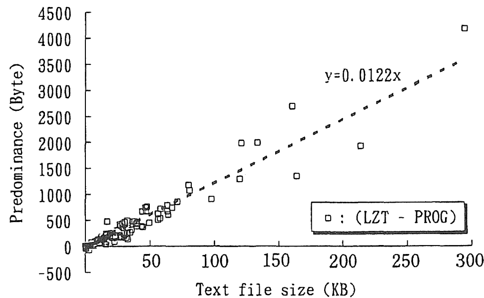

図9 PROGのLZTに対する優位性

表2 提案法による圧縮率の改善

対	テキストファイル	実行ファイル
FGK	28.2 %	19.3 %
LZT	1.22 %	1.03 %

圧縮時が，テキストファイルよりも実行ファイルがそれぞれ若干ではあるが余分に処理時間を要している．

FGKの実行時間の大半を占めるのはハフマン木の更新過程である．単一文字を対象とするFGKに対し文字列単位の圧縮を考慮した提案法ではハフマン木の更新回数が減少する．その結果，FGKと比較して大幅に実行時間を短縮した．しかし，一方のLZTではハフマン木自体が必要ではない．よって，LZTと比較すると提案法はハフマン木の更新分だけ処理負担が増大し，その分だけ実行時間も増加する．

5. おわりに

多くのデータに共通する冗長性・局所性に対処可能な動的辞書法の一部に動的ハフマン符号を同期させた1パスデータ圧縮アルゴリズムを提案した．辞書サイズを4Kバイトとして実装した提案法は符号化対象がバイト単位である動的ハフマン法や文字列単位である動的辞書法と比較して圧縮率の面での改善を実現した．

表3 50KB当たりの圧縮および復元時間（秒）

圧縮方法	テキストファイル		実行ファイル	
	圧縮時	復元時	圧縮時	復元時
FGK	13.396	12.702	15.662	14.897
LZT	6.472	6.271	7.571	7.022
PROG	7.998	7.784	12.068	11.418

実行時間もわずかな増加に留まった．圧縮率と実行時間の関係は互いに相反するものである．この意味で提案法は両者を比較的バランスさせた良いデータ圧縮法であるといえる．

謝　辞

本研究は財団法人 日東学術振興財団の第14回（平成9年度）助成により達成された．ここに謝意を表する．

参考文献

1) D.A.Lelewer and D.S.Hirschberg: "Data Compression," *ACM Computing Surveys*, Vol.19, No.3, pp.261–296, 1987.

2) T.C.Bell, J.G.Cleary and I.H.Witten: Text Compression, Prentice Hall, New Jersey, 1990.

3) T.C.Bell: "Better OPM/L Text Compression," *IEEE Trans. on Commun.*, Vol.34, No.12, pp.1176–1182, 1986.

4) T.A.Welch: "A Technique for High-performance Data Compression," *IEEE Trans. on Computer*, Vol.17, No.6, pp.8–19, 1984.

5) 奥村晴彦: 「C言語による最新アルゴリズム辞典」, 技術評論社, 1991.

6) P.Tischer: "A Modified Lempel-Ziv-Welch Data Compression Scheme," *Australian Computer Sci. Commun.*, Vol.9, No.1, pp.262–272, 1987.

7) 植松友彦:「文書データ圧縮アルゴリズム入門」, CQ出版社, 1994.

8) D.E.Knuth: "Dynamic Huffman Coding," *J. of Algorithms*, Vol.6, No.2, pp.163–180, 1985.

9) J.S.Vitter: "Dynamic Huffman Coding," *ACM Trans. on Mathematical Software*, Vol.15, No.2, pp.158–167, 1989.

（受理　平成10年3月20日）