トライと三分木を使った高速化技法
LZ78符号によるファイルの圧縮と改良
後編

広井 誠

前編（2006年11月号，pp.151-160）はLZW符号のアルゴリズムと辞書を表すデータ構造のトライについて詳しく説明しました．そして，トライの実装にTernary Search Tree（TST）を用いることで，符号化の処理速度を高速にすることができました．後編では，辞書が満杯になったときの対処方法について説明します．

LZW符号の改良

● 辞書が満杯になったときの処理が要改善点

前編で作成したプログラムは，辞書が満杯になった時点で辞書を固定し，あとはその辞書をそのまま使ってファイルを圧縮しました．ところが，この方法では不都合な場合もあります．たとえば，データの種類が異なる複数のファイルをアーカイバでまとめてから圧縮することを考えてみましょう．途中で辞書が満杯になりデータの種類が変化すると，それ以降のデータはほとんど圧縮することができません．

一例として，Canterbury Corpus[6]で配布されているThe Canterbury Corpusをアーカイバtarでまとめ（canterbury.tar），LZW符号で圧縮した結果を**表1**に示します．

このように辞書サイズが小さいときは，ファイルを圧縮することはほとんどできません．この場合は大域的な辞書を作成していることが裏目となり，データの変化についていけないのです．このような問題を解決するために，辞書が満杯になったときの対処方法がいくつか提案されています．本稿では，二つの方法を取り上げます．一つは辞書が満杯になったらクリアする「辞書クリア方式」，もう一つは辞書から不要な語を取り除いていく「LZT符号」です．

辞書クリア方式

● 満杯になった辞書を初期化する

辞書をクリアする方法は，LZW符号のプログラムを修正するだけで簡単に実現できます．辞書が満杯になったら辞書を再初期化するだけで，あとは今までと同じように記号を読み込んで辞書に登録していくだけです．符号化のプログラムを**リスト1**に示します．

辞書のチェックはグローバル変数code_countとnode_maxを使います．code_countは，記号を符号化した回数と復号した回数を，node_maxは辞書番号の最大値を表します．code_countは256に初期化するので，node_max以上になったら辞書が満杯になったとわかります．このとき，関数

表1　LZW符号によるcanterbury.tarの評価結果

PROG	サイズ	辞書サイズ		
		8K	32K	128K
LZW	2,821,120	2,407,702	2,643,586	1,174,612

リスト1　符号化（辞書クリア方式）

```
void encode( FILE *fp )
{
  int p;
  /* 最初の文字を読み込む */
  if( (p = fgetc( fp )) == EOF ) return;
  /* トライの初期化 */
  init_trie();
  /* トライをたどる */
  for(;;){
    int c, q;
    if( (c = fgetc( fp )) == EOF ){
      /* 一致した位置までを出力 */
      cbt_encode( p );
      break;
    }
    if( (q = search_code( p, c )) == NIL ){
      /* p が最長一致位置 */
      cbt_encode( p );
      /* 辞書の更新 */
      add_leaf( p, c );
      /* 不一致文字を先頭文字にする */
      p = c;
      /* 辞書が満杯かチェックする */
      if( code_count >= node_max ) dic_clear();
    } else {
      p = q;
    }
  }
}
```

dic_clear()を呼び出して辞書を初期化します．

次は復号のプログラムを**リスト2**に示します．

復号の場合，変数qの節に記号を追加しますが，辞書をクリアした直後は記号を追加することはできません．そこで変数flagを用意して，辞書をクリアしたときflagをTRUEにセットします．そして，flagがFALSEのときだけqに記号を追加します．

最後に，辞書をクリアする関数dic_clear()を**リスト3**に示します．

トライの初期化処理とほとんど同じです．グローバル変数node_count，code_count，code_bitsの初期化を忘れないように注意してください．プログラムの修正はこれだけです．

LZT符号

● 長い間使われていない語を辞書から追い出す

次はLZT符号について説明します．LZT符号は1987年にTischer氏によって開発されました．おもな改良点は，辞書が満杯になった場合，長い間使われていない（最長時間未使用：Least Recently Used）語を取り除くことで辞書の空きスペースを作るところです．このような操作を「LRUスキーム」と呼びます．

LZT符号はLRUスキームを行うために符号化と復号に時間がかかりますが，少ないメモリでも高い圧縮率を期待できます．また，データの局所的な偏在も辞書に反映できます．

辞書の管理は「キュー（queue）」を使って行うと簡単です．キューは先入れ先出し（FIFO）のデータ構造です．トライをたどっていくときに，通過した節をキューから取り出して最後尾へ移動します．そうすると，アクセスされない節ほどキューの先頭に集まることになるので，節を削除する場合はキューの先頭から行えばいいわけです．

一般に，キューはリング・バッファや連結リストを使うと簡単に実装できます．データの出し入れがキューの先頭と最後尾だけなら，これらのデータ構造で十分なのですが，キューの途中からデータを削除する場合に適しているとはいえません．

リング・バッファは配列を使って実装するので，削除した分だけデータを移動しないといけません．また，連結リストでデータを削除する場合，そのデータを保持しているセルと，一つ手前のセルの両方が必要になるため，リストの先頭から順番にセルをたどらないといけません．どちらの方法でも，キューの途中からデータを削除するには時間がかかります．

そこで，本稿では「双方向リスト」を使って，節の順番を管理することにします．まず最初に，基本的な事項ですが，双方向リストについて簡単に説明します．

● 双方向リスト

名前が示すように，双方向リストとは直後のセルだけでなく直前のセルへのポインタをもたせたデータ構造です．双方向リストは「重連結リスト（doubly-linked list）」と呼ばれることもあります．**図1**を見てください．

連結リストは後ろ方向にしかセルをたどることができませんが，双方向リストなら前後どちらの方向へもセルをたどることができます．セルを削除する場合も，前後のセルがわかるので簡単に削除できます．この双方向リストはメモリ管理などでよく使われるデータ構造です．

実際にポインタを使って双方向リストを定義すると，**リスト4**のようになります．ここでは格納するデータを整数値としました．

prevが直前のセルへのポインタを格納し，nextが直後のセ

リスト2　復号（辞書クリア方式）

```
void decode( FILE *fp )
{
  int q, len, flag = FALSE;
  int size = header_read();
  if( size == 0 ) return;
  /* トライの初期化 */
  init_trie();
  /* 最初の復号 */
  q = cbt_decode();
  len = output_code( q, fp );
  size -= len;
  /* トライをたどって復号 */
  while( size > 0 ){
    int c = cbt_decode();
    if( c < node_count ){
      len = output_code( c, fp );
      if( flag ){
        flag = FALSE;
      } else {
        add_leaf( q, buff[len - 1] );
      }
    } else {
      /* 辞書範囲外の場合 */
      add_leaf( q, buff[len - 1] );
      len = output_code( c, fp );
    }
    size -= len;
    q = c;
    /* 辞書が満杯かチェックする */
    if( code_count >= node_max ){
      dic_clear();
      flag = TRUE;
    }
  }
}
```

リスト3　辞書のクリア

```
void dic_clear( void )
{
  int i;
  /* 0 - 255 は最初から登録しておく（ハッシュ表に入れない） */
  for( i = 0; i < 256; i++ ){
    trie[i].code = i;
    trie[i].parent = NIL;
    trie[i].left = trie[i].middle = trie[i].right = NIL;
  }
  for( i = 256; i < node_max; i++ ){
    trie[i].parent = NIL;
    trie[i].left = trie[i].middle = trie[i].right = NIL;
  }
  node_count = 256;
  code_count = 256;
  code_bits  = 9;
}
```