# 圧縮アルゴリズムBWTの性能確認実験

**An Experimental Study of Compression Algorithms**

http://www.tani.cs.chs.nihon-u.ac.jp/g-2005/pandora/

**谷 研究室　松田 賢幸**

**Takayuki MATSUDA**

**概要**

今回は圧縮技術のうちのひとつであり，比較的優れていると言われる BWT 圧縮のアルゴリズムを紹介し，その性能確認実験を行う．

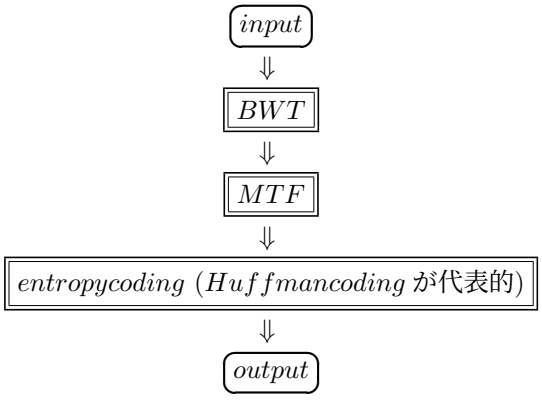
## 1　はじめに

小規模な記憶領域にデータを格納する場合や，低速な回線でデータの送受信を行なう場合，圧縮技術は未だに有用である．また，最近では，データ検索のために圧縮技術を応用する研究もされており，今まで以上に圧縮技術の必要性は高まっている印象を受ける．今回はそのような圧縮技術の中で近年注目されている圧縮法を紹介し，その性能確認を行う．

$'.bz'$,$'.bz2'$ といた拡張子は $bzip$ または $bzip2$ という圧縮ツールによって圧縮されたファイルのものである．この $bzip$ は比較的高性能と言われている．現に $Linux$ 系 $OS$ のソース配布などでは，今まで良く使用されてきた $gzip$(拡張子は $'.gz'$) から，$bzip$ へと圧縮法を変える動きもある．今回紹介する $Burrows-WheelerTransform$(以下 $BWT$) 圧縮は，そのような $bzip$ で使用されているアルゴリズムである．その $BWT$ 圧縮がどのように高い圧縮率を実現しているかを以下で述べていきたいと思う．

## 2　BWT圧縮の流れ

先に述べたような基本的な $BWT$ 圧縮は次のような流れになる．




以下では $BWT$,$Movetofront$(以下 $MTF$),$Huffmancoding$ のアルゴリズムを個別に紹介する．

## 3　BWTアルゴリズムの概要

$BWT$ アルゴリズムは入力文字列を圧縮のしやすい形に変え，出力するアルゴリズムである. その手順を簡単に紹介すると以下のようになる．

### 3.1　符号法

1. 入力文字列の末尾に特異な終了記号を加えた文字列を生成する．
2. 1. で生成された文字列を一文字づつ巡回シフトさせ, そのような文字列を並べた (概念上の) 行列を生成する．
3. 2. で並べられた文字列を辞書順に並べ換える．
4. 3. の文字列それぞれの最後の文字を順に出力する．

具体例を以下に示す.
入力文字列を $mississippi$ とする．

1. 入力文字列の末尾に特異な記号 @ を加える．
   $mississippi@$
2. 1. の文字列を巡回シフトさせ作った行列は以下のものになる．

$$\begin{pmatrix} mississippi@ \\ ississippi@m \\ ssissippi@mi \\ sissippi@mis \\ issippi@miss \\ ssippi@missi \\ sippi@missis \\ ippi@mississ \\ ppi@mississi \\ pi@mississip \\ i@mississipp \\ @mississippi \end{pmatrix} \tag{1}$$

3. 2. で出来た行列を辞書順に並べ替えたものが以下の行列である．また，このとき特異な記号は他のどの文字よりも先にくるようにする．

$$\begin{pmatrix} @mississippi \\ i@mississipp \\ ippi@mississ \\ issippi@miss \\ ississippi@m \\ mississippi@ \\ pi@mississip \\ ppi@mississi \\ sippi@missis \\ sissippi@mis \\ ssippi@missi \\ ssissippi@mi \end{pmatrix} \tag{2}$$

4. 3. で出来た行列の末尾を縦に取り出したものが出力となる．
$output = ipssm@pissii$

## 3.2 復号法

$BWT$ の復号法について述べる．抽象的な言葉だと解かりづらいので，具体例を用いて話を進める．
対象文字列を $ipssm@pissii$ とする．

1. 符合法より対象文字列から以下のような虫食い状態の二次配列がわかる (ここで『?』は不明文字)．

$$\begin{pmatrix} ??????????i \\ ??????????p \\ ??????????s \\ ??????????s \\ ??????????m \\ ??????????@ \\ ??????????p \\ ??????????i \\ ??????????s \\ ??????????s \\ ??????????i \\ ??????????i \end{pmatrix} \tag{3}$$

2. さらに先頭の文字はソーティングされていることから．

$$\begin{pmatrix} @??????????i \\ i??????????p \\ i??????????s \\ i??????????s \\ i??????????m \\ m??????????@ \\ p??????????p \\ p??????????i \\ s??????????s \\ s??????????s \\ s??????????i \\ s??????????i \end{pmatrix} \tag{4}$$

3. 巡回シフトさせたことより 2. から各文字の次に来る文字が解かった (例：$i$ の次に来る文字は @)．また，末尾に特異な記号を加えたことより，最後に @ のある列が元の文字列であることが解かる．それらの情報をもとに，二次配列を復元する．

   今回の例では 6 列目が元の文字列であり，$m$ の次の文字は $i$ なので，以下のように 6 列目を復元対象として，$i$ を $m$ の次に付け足す．

$$\begin{pmatrix} @??????????i \\ i??????????p \\ i??????????s \\ i??????????s \\ i??????????m \\ mi?????????@ \\ p??????????p \\ p??????????i \\ s??????????s \\ s??????????s \\ s??????????i \\ s??????????i \end{pmatrix} \tag{5}$$

4. さて，次の文字だが，$i$ という文字が複数ある．こういった場合は上から何番目の $i$ であるかを調べる必要がある．5 列目を見てほしい．この列から $m$ の次にくる文字が $i$ だと解かった．そして，この列先頭にある $i$ は上から数えて 4 番目の $i$ である．よって，最後の行の上から 4 番目の $i$ の次に来る文字を見ればいい．これでは 12 列目にあたる．

つまり、$s$ が次に来る文字だと解かるのである.

$$\begin{pmatrix} @??????????i \\ i??????????p \\ i??????????s \\ i??????????s \\ i??????????m \\ mis????????@ \\ p??????????p \\ p??????????i \\ s??????????s \\ s??????????s \\ s??????????i \\ s??????????i \end{pmatrix} \tag{6}$$

5. 以下，同じ操作を繰り返していく.

$$\begin{pmatrix} @??????????i \\ i??????????p \\ i??????????s \\ i??????????s \\ i??????????m \\ mississippi@ \\ p??????????p \\ p??????????i \\ s??????????s \\ s??????????s \\ s??????????i \\ s??????????i \end{pmatrix} \tag{7}$$

6. 上記の二次配列から元の文字列は $mississippi@$ だと解かった.

# 4 MTF アルゴリズムの概要

$MTF$ アルゴリズムは入力文字列に対し，出力が偏ったものになりやすいという特性を持つ．その手順を以下で示す.

## 4.1 符号法

1. 入力文字列から一文字読み込む.
2. 1. で読み込んだ文字がリストの中にあれば，そのリスト内の先頭から何番目かを示す番号を出力し，その文字をリストの先頭にもってくる．なければそのまま文字を出力し，リストの先頭にその文字を加える．(通常，初期状態のリストは空)
3. 文字列の最後まで 1., 2. を繰り返す.

基本的に上の手順になるが，$BWT$ 圧縮で $MTF$ を用いるときはリスト内に最初から全ての文字が含まれているものとする.

さらに具体例を以下に示す.

入力文字列を $ipssm@pissii$ とする.

入力
$ipssm@pissii$

| 出力 | リスト |
|---|---|
| 0 | $[i, p, s, m, @]$ |
| 01 | $[p, i, s, m, @]$ |
| 012 | $[s, p, i, m, @]$ |
| 0120 | $[s, p, i, m, @]$ |
| 01203 | $[m, s, p, i, @]$ |
| 012034 | $[@, m, s, p, i]$ |
| 0120343 | $[p, @, m, s, i]$ |
| 01203434 | $[i, p, @, m, s]$ |
| 012034344 | $[s, i, p, @, m]$ |
| 0120343440 | $[s, i, p, @, m]$ |
| 01203434401 | $[i, s, p, @, m]$ |
| 012034344010 | $[i, s, p, @, m]$ |

出力文字列: 012034344010

## 4.2 復号法

MTF アルゴリズムの復号法は，符号法の際に最初に作ったリストと，出力文字列によって行われる．操作自体は符号法とさして変わりはない.

下記のように数字を見て，その数字が指す番号の文字を出力していけば，復号できる．また，見て解かるとおり，符号法と同じように，リストは更新してゆく.

出力文字列が 012034344010 であったとする.

入力
012034344010

| 出力 | リスト |
|---|---|
| $i$ | $[i, p, s, m, @]$ |
| $ip$ | $[p, i, s, m, @]$ |
| $ips$ | $[s, p, i, m, @]$ |
| $ipss$ | $[s, p, i, m, @]$ |
| $ipssm$ | $[m, s, p, i, @]$ |
| $ipssm@$ | $[@, m, s, p, i]$ |
| $ipssm@p$ | $[p, @, m, s, i]$ |
| $ipssm@pi$ | $[i, p, @, m, s]$ |
| $ipssm@pis$ | $[s, i, p, @, m]$ |
| $ipssm@piss$ | $[s, i, p, @, m]$ |
| $ipssm@pissi$ | $[i, s, p, @, m]$ |
| $ipssm@pissii$ | $[i, s, p, @, m]$ |

元の文字列: $ipssm@pissii$

# 5 Haffmancoding の概要

$Huffmancoding$ は文字列内に出てくる文字の出現頻度を調べ，出現頻度が高いものには短い符号を，低いものには長い符号を割り当てるという考え方で作られたアルゴリズムである．

## 5.1 符号法

1. 入力文字列を読み込み，一文字毎の出現頻度を調べ，頻度表を作る．

2. 頻度表の中から出現頻度の少ないものを 2 つ見つける．

3. 見つけた 2 つの文字にそれぞれ 0 と 1 を割り当てる．

4. 選んだ 2 つの文字をひとまとめにし，その頻度は 2 つの文字の出現頻度を足し合わせたものとする．

5. 頻度表の中の文字が全てひとまとまりになるまで 2., 3., 4. を繰り返す．

6. 文字列をそれぞれの文字に対応した符号に置き換える．

具体例を示す．
入力文字列を 012034344010 とする．

1. 入力文字列をもとに頻度表を作ると以下のようになる．

| 文字 | 頻度 |
|---|---|
| 0 | 4 |
| 1 | 2 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |

文字への符号の割り当てを行う．

2. 上の表で一番頻度が少ないのは 2, 1(1 でも 3 でも良い) の 2 つなので，

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 4 | |
| 1 | 2 | 0 |
| 2 | 1 | 1 |
| 3 | 2 | |
| 4 | 3 | |

3. 上のように 1 に 0,2 に 1 を割り当てる．

4. 更に頻度表の更新を行う．

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 4 | |
| 1 | 3 | 0 |
| 2 | | 1 |
| 3 | 2 | |
| 4 | 3 | |

5. 以下同様に，

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 4 | |
| 1 | 3 | 10 |
| 2 | | 11 |
| 3 | 2 | 0 |
| 4 | 3 | |

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 4 | |
| 1 | 5 | 10 |
| 2 | | 11 |
| 3 | | 0 |
| 4 | 3 | |

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 4 | 0 |
| 1 | | 10 |
| 2 | 5 | 11 |
| 3 | | 0 |
| 4 | 3 | 1 |

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 7 | 0 |
| 4 | | 1 |
| 1 | | 10 |
| 2 | 5 | 11 |
| 3 | | 0 |

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 7 | 00 |
| 4 | | 01 |
| 1 | | 110 |
| 2 | 5 | 111 |
| 3 | | 10 |

| 文字 | 頻度 | 符号 |
|---|---|---|
| 0 | 12 | 00 |
| 4 | | 01 |
| 1 | | 110 |
| 2 | | 111 |
| 3 | | 10 |

6. 頻度表がひとまとまりになったので，文字に対応している符号へ入力文字列を置き換える．

   012034344010

   ⇒ 00/110/111/00/10/01/10/01/01/00/110/00

   ⇒ 00110111001001100101001100 0

   これでは長くなったのではないかと思う人もあるかもしれないが，文字列を *bit* 列に置き換えたものと考えてほしい．例えば 1 文字が 1*byte* の文字コード表されていたとすると，例にだした入力文字列は $12 \times 8 = 96bit$ となるが置き換えられた後では，$27bit$ で表すことができたのである．

### 5.2 復号法

出力文字列と頻度表から復号を行なう．

復号は出力文字列を先頭から見ていき，頻度表の中の符合と一致する文字列が表れたら文字を置き換えていけばよい．

具体例を以下に示す．

ここで出力文字列は 00110111001001100101001100 0
頻度表は

| 文字 | 符号 |
|---|---|
| $A$ | 00 |
| $E$ | 01 |
| $B$ | 110 |
| $C$ | 111 |
| $D$ | 10 |

とする．

00110111001001100101001100 0 を先頭から見ていくと，まず 0 を見ることになる．頻度表の符合の中に 0 は無いため，さらに一文字先を見る．すると 00 という文字列を見ることになる．頻度表を見ると 00 という符合は $A$ に対応していることがわかる．なので，00 を $A$ に置き換える．すると，$A$110111001001100101001100 0 となる．同じように次の文字を読み込むと，1 を見ることになり，頻度表の中にないので，さらに先を見ていくと，110 が見つかる．これは $B$ に対応する符合なので，これを $B$ に置き換える．以下同様に作業を進めていくと $ABCADEDEEABA$ という文字列が元の文字列であったことがわかる．(ここでは説明を解り易くするため，頻度表を符合化の際に作ったものと変えてある．そのため符合前の文字列と異っていることに注意)

動きを簡略に示すと以下のようになる．

$$
\begin{aligned}
&0011011100100110010100110000 \\
\Rightarrow\ &0/011011100100110010100110000 \\
\Rightarrow\ &00/11011100100110010100110000 \\
\Rightarrow\ &A/1101110010011001010011000 \\
\Rightarrow\ &A/1/101110010011001010011000 \\
\Rightarrow\ &A/11/01110010011001010011000 \\
\Rightarrow\ &A/110/11100100110010100110000 \\
\Rightarrow\ &AB/1110010011001010011000 \\
\Rightarrow\ &AB/1/110010011001010011000 \\
\Rightarrow\ &AB/11/10010011001010011000 \\
\Rightarrow\ &AB/111/0010011001010011000 \\
\Rightarrow\ &ABC/001001100101001100 0 \\
&\ \ \vdots \\
\Rightarrow\ &ABCADEDEEABA
\end{aligned}
$$

ちなみに今回の $Huffmancoding$ の説明は，簡略化してあるため本質的に正しくない表現が混じっていることを付け加えておく．

## 6 BWT 圧縮の利点と欠点

上のようなアルゴリズムで実現されている $BWT$ 圧縮の特徴をここで述べる．$BWT$ 圧縮の本質は，その名が

示すとおり $BWT$ アルゴリズムにある．$BWT$ アルゴリズムが圧縮しやすい変換を成すことで汎用性をもち，多くのデータ形式で他の圧縮よりも高い圧縮率を誇るのである．ただ、ソートの処理時間がデータ量に指数比例してしまうため、あらかじめデータの処理の上限を決める必要がある．($bzip$ では $900kbyte$ が上限となっている) このために，ファイルサイズが大きければ圧縮率が良くなる保証はない．

# 7 確認実験の結果

- 実験方法

  いくつかのデータを 3 種類の圧縮法で圧縮し，その平均の圧縮率，圧縮時間，解凍時間を比較する．3 種類の圧縮法とは以下のものである．

  1. $BWT + MTF + Huffman$($BWT$ 圧縮)
  2. $MTF + Huffman$
  3. $Huffman$

  また，このときの圧縮率とは元のデータサイズに対する圧縮ファイルのサイズとする．よって圧縮率は以下のようになる．

  圧縮後のファイルサイズ:$after$

  元のデータサイズ:$before$

  圧縮率:$compress$

  $compress = (after/before) * 100$

- 実験結果

  元のデータサイズの平均: 14358byte

| | 平均圧縮率 | 平均圧縮時間 | 平均解凍時間 |
|---|---|---|---|
| 1 | 51.81 | 2.48 | 3.18 |
| 2 | 79.59 | 0.34 | 0.27 |
| 3 | 72.02 | 0.26 | 0.2 |

  時間の単位:秒

  圧縮率の単位:%

- 考察

  なにより驚いたのは $MTF + Huffman$ の組み合わせが $Huffman$ 単体で動かしたときよりも平均圧縮率が悪くなったという点である．当たり前のことだが，アルゴリズムは単純に組み合わせただけでは良い結果を得られないという一例だろう．
  また，$BWT + HTF + Huffman$ の圧縮率が他の方法とかなりの差をつけているところにも注目したい．これは，$BWT \Rightarrow MTF$ という変換が理論上だけでなく，実際に有効だという証明となっている．
  しかし，$BWT + MTF + Huffman$ は実行時間で他から大きく遅れをとっている．これについては大方の予想がついていたが，$BWT$ を除いただけの $MTF + Huffman$ との差がこれほどつくとは思わなかった．
  今回の実験で $BWT$ の持つ特性が，一般に言われているものと特に変わらないことが確認できた．

# 参考文献

[1] Giovanni Manzini
The Burrows-Wheeler Transform:Theory and Practice

[2] Sebastian Deorowicz
Second step algorithms in the Burrows-Wheeler compression algorithm

[3] Paolo Ferragina and Giovanni Manzizi
An experimental study of a compressed index