5S−2

キーワード表を用いたレンペル・ジブ圧縮(LZK)

金 時哲 三浦 孝夫

産能大学

1 前書き

レンペル・ジブ符号化(LZ)は、文字列などを符号化する方法に多く用いられている。この方法は1パス方式であり、ファイルを読みながら圧縮を行うための辞書を動的に生成する。しかし、LZ方法では対象となるデータについての知識を有さず、時として圧縮効率が良くないことがある。この研究の目的はLZ方法の特徴を生かした符号化方法を工夫することである。

2 レンペル・ジブ符号化(Lempel・Ziv coding)

LZ符号化の特徴は、ASCIIコードと辞書を用いることによって、頻繁に現れる文字列に対して短い符号を与えることである。この方法は、最初にASCIIコードを辞書に登録しておいて、ファイルから1バイトずつ読み、辞書とのパターンマッチングを行って、辞書に登録されていなければこれを追加し、そのあと同じ文字列が出現するとその文字列の符号ビットを出力することによって短いビットを与える。

この符号化圧縮の利点は、頻繁に現れる文字列に対して、短いビットを与えること、ファイルを1回しか読まないこと、表をもたないことにある。反面、頻繁に現れる文字列をつくるため、多くの文字列の組み合わせが必要となるので、辞書の中には無駄な領域を多く持つ傾向がつよい。これは、同じ文字列が少数であればあるほど無駄な領域を持つ確率は高くなる(例:全角文字のファイル)。

3 キーワード表を用いたレンペル・ジブ符号化(LZK)

ここで提案するLZK符号化の特徴は、本質的にはハフマン符号化方法とLZ方法を組み合わせるものである。1バイトで表現できる256個のコードを128個の文字コードだけにして、後半の128個には1ファイルの中で最も頻度が高い文字列を代入することである。

従って、最も頻度が高い文字列128個を先に代入しておくことにより、その文字列を生成するための処理を単純化することができる。最も頻度が高い文字列にもっと短い符号を与えることことによって((文字列の数(128個)×文字列の総バイト))−コードの数(128バイト)分なみのデータが節約できる。二度目にファイルを読むと、先にコード化された文字列の前後に現れやすいパターンがでてくる。これを辞書の中にまた登録することによって、もっと短い符号を与えることができる。しかし、LZ符号化は1パスで実行できるが、LZK符号化では2パスであるので、遅くなる。またLZK符号化は文字コードしか使わないので、一般のビットイメージそのままでは符号化されない。この全角文字列を区別させるために全角文字の始まりと終わりに符号を与える必要がある。

[1]Keyword table used Lemple-Ziv compression Sichull Kim SANNO College

4 LZK符号化の実現

ここでは簡単にLZK方式によるアルゴリズムの説明をする。まず128個の文字列を代入し、ソート、表の出力、符号化を行う。そして復号化では、表を読んで辞書に登録し、復号化を行う。

```
{
文字コード(128個)を辞書に登録辞書サイズ(4096,8188...);
while((c = getc(1バイト読み込み))が終わりでなければ)
    if(isalpha(c)) アルファベットであれば
アルファベット(52個)のみでつながる文字列をtrieに代入;
文字列の頻度順にソートをかける;
for(高い頻度128個を辞書(128 〜 255)に代入する.
128個の文字列を出力する;
while(c = getc(ファイルを始めから読む)){
    s = c;
    while(sが空でなければ繰り返す) {
string = s;
matching[貯める] = c;
c = getc(読み込み);
s = child(sの子供がcか空か);
    }
    string(辞書のコードを出力);
    辞書(256 〜 )にstring+cの文字列を登録する;
  }
}
```

LZKの復号化のときは、ファイルを2回読む必要がない。それは符号化のときに128個の文字列をファイルの中に入れたのである。従って、符号化されたファイルの先頭から128個の文字列を読めばいいのである。

```
{
ASCIIコード(128個)を辞書に登録する;
for(ファイルから128個の文字列を読む)
    辞書(128 〜 255)に登録する;
c = getc(次の位置からファイルを読む);
while(ファイルの終わりでなければ){
    s = c;
    do { string = s;
matching[貯める] = c;
c = getc(読み込み);
s = child(sの子供がcか空か);
    } while(sが空でなければ)
    string(辞書のコードにある文字列を出力する);
    辞書(256 〜 )にstring+cの文字列を登録する;
  }
}
```

5 LZ 符号化と LZK 符号化との比較

ここでは、以下のような普通のプログラムの文字列を取り出し、符号化されたときの文字数と辞書の領域の活用についての比較を行う。

（例）

"void swap(int *p, int *q)"

辞書	LZ 符号化		LZK 符号化	
コード	文字列	出　力	文字列	出　力
0	0	0	0	
⋮	⋮	⋮	⋮	⋮
128			void	
129			int	
⋮	⋮	...	vdots	
255	255	255	abc	
256	vo	v	128 □	128
257	oi	o	□s	□
258	id	i	sw	s
259	d □	d	wa	w
260		□	ap	a
261	sw	s	p(	p
262	wa	w	(129	(
263	ap	a	129 □	129
264	p(	p	□*	□
265	(i	(	*p	*
266	in	i	p,	p
267	nt	n	, □	,
268	t □	t	□263	□
269	□*	□	263*	263
270	*p	*	*q	*
271	p,	p	q)	q
272	, □	,	)	)
273	□266	□		
274	266268	266		
275	268*	268		
276	*q	*		
277	q)	q		
278	)	)		

上の表で示す例から LZ 符号化と LZK 符号化に同じファイルを与えて、その変化の違いを捜し出す。その前に LZK 符号化にはファイルを読んで作られたキーワード表が辞書に登録されているとする。この例からいくつかの特徴を見出すことが出来る。

第1に辞書の領域の長さに注意する。同じファイルに対し辞書の領域が大きいと言うのは、辞書の中に無駄な文字列が多く含まれているといえる。

第2の点は、出力された語の回数である。LZ 符号化は 23 回出力されて、LZK 符号化では 17 回しか出力されていない。プログラムなどには同じ文字列が多く現れる可能性が高い。それに、現れる可能性が高い語の前後の組合せも多く現れる可能性が高いので、LZ 符号化よりは、LZK 符号化の方が圧縮率が高くなると言える。

しかし、LZK 符号化はファイルから読んだ文字列とキーワード表とのパターンマッチングを行うために、ファイルを 2 回読みすることから LZ 符号化方法よりは遅い。

6 結び

このLZK 符号化には大きな欠点が2つある。1つは、2回読みするので符号化する時間が遅い。また文字コードのみにしたのでビットイメージは符号化できない。今後の課題として、符号化する時間を早くすることである。キーワード表の知識を活用する方法なども考えられる。

参考文献

[1] James A. Storer, "DATA COMPRESSION methods and theory" *COMPUTER SCIENCE PRESS*, 1988.

[2] A.V.Aho,J.E.Hopcroft,J.D.Ullman, "データ構造とアルゴリズム", 培風館, 1990.

[3] 瀧　保夫, "情報論", 岩波出版, 1990.