RLE - Run Length Encoding

Student IONUȚ SILVIU NICULESCU
Student CĂTĂLIN BOJA
Student ALEXANDRU STANCIU
Student IONUȚ BULUMACU
Faculty of Economic Cybernetics, Statistics and Informatics,
Academy of Economic Studies

1. Principle of RLE

Run-**l**ength **e**ncoding (RLE) is a very simple form of data compression encoding. It is based on simple principle of encoding data. This intuitive principle works best on certain data types in which sequences of repeated data values are noticed; RLE is usually applied to the files that contain a large number of consecutive occurrences of the same byte pattern.

In data compression are used many algorithms, some are simple ones and others are complex algorithms. Their efficiency depends on what type of data is being compressed. The easiest algorithm used today widely is the RLE.

The basic RLE principle is to detect sequences of repeated data values and after that to replace this sequence with two elements:

- **the number of the same characters**
- **the character itself**

The encoding process is effective only if there are sequences of 3 or more repeating characters. For example:

ASCII

after RLE compression it becomes :

ASC2I

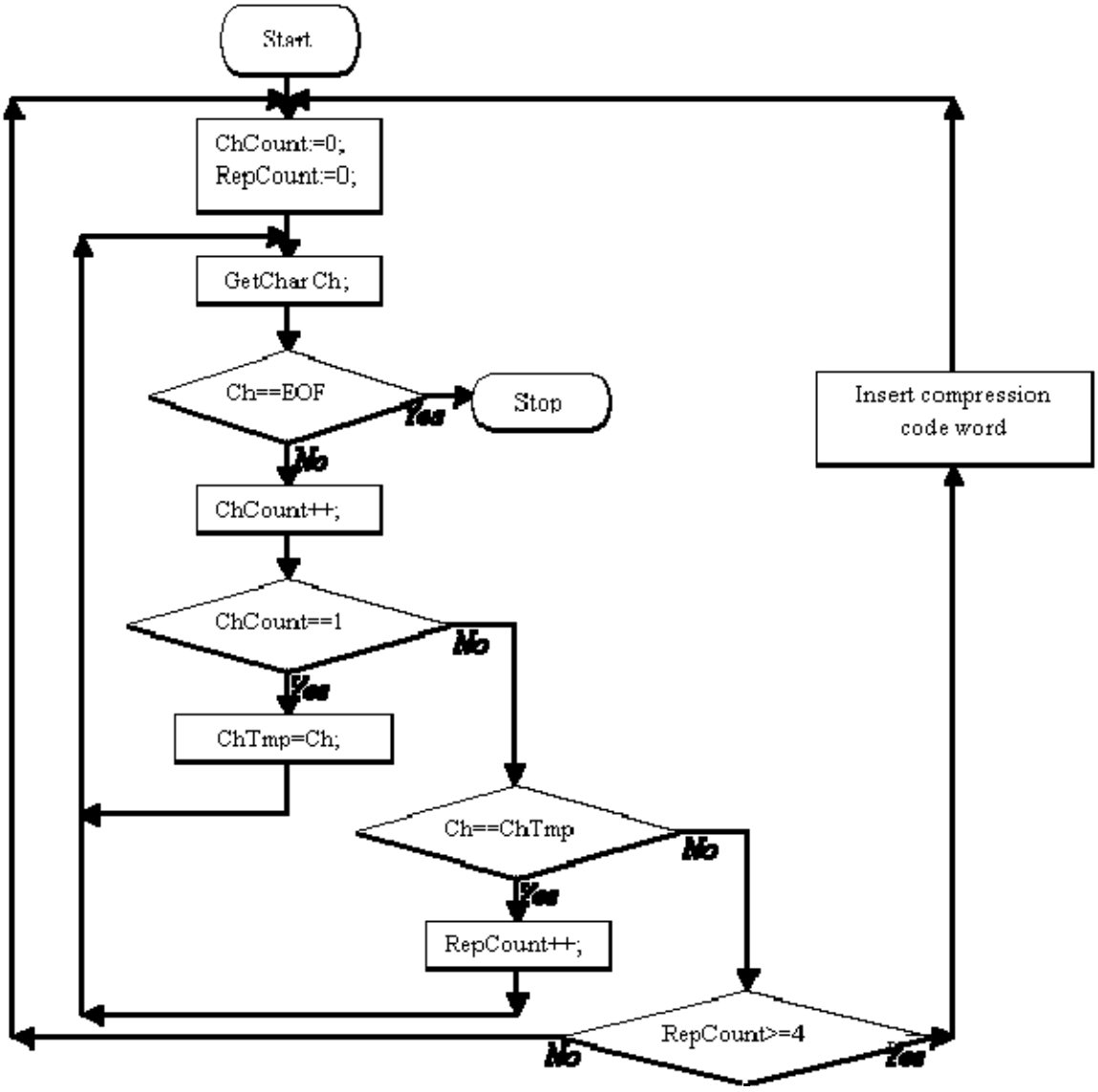
As you can see, in the above example we don't have a real compression, because when we are decoding we will do it wrong. To distinguish the replace sequence from the data that hadn't been coded, we use a new symbol that represents a **control character**. This special character, in the decoding process, will be the start point of a sequence of compressed data. So the replacing sequence of 3 or more repeating characters will have 3 characters:

- **the special character**;
- **the number of repetition**;
- **the repeating character**.

Fig 1. Format of three byte code word

Where :

CTRL - control character which is used to indicate compression
COUNT- number of counted characters in stream of the same characters
CHAR - repeating characters.



Process of RLE starts with initialisation of character counter, repetition counter and a variable, which represents the current character (Ch), then if all characters in file have been processed encoding ends. If there are more characters then Ch variable is being stored in temporary variable (if ChCount equals 1), else actual character is being compared to the previous character and then result of that comparison leads to repetition counter increment or to another comparison in which it is being tested if the number of consecutive characters is greater than four in other words does the stream is just copied or coded according to code word shown in Fig 1.

For example the stream:

aaaabbbaabbbbbcccccccdabcbaaabbbbcccd

with the control character #, it becomes :

#4a#3baa#5b#8cdabcb#3a#4b#3cd

When the character used as control character is in the stream that must to be compressed, it will be replaced by the next sequence:

- **special character**;
- **0 value, because normally here it was the number of repetition**
- **1 or the special character**.

If a sequence of repeating control characters exists in the file that must to be compressed, it will be coded with :

- **special character**;
- **1, because it is an unused value in the compression process.**
- **the repetition number of special character**;

2. Examples of RLE Implementations

RLE algorithms are parts of various image compression techniques like BMP, PCX, TIFF, and is also used in PDF file format, but RLE also exists as separate compression technique and file format.

2.1 CompuServe standard for RLE file format

CompuServe RLE file format standard was made in the 80's and defines the compression for 1-bit images.

Header sequence in RLE file represents Graphic Mode Control, control is initiated when program runs onto a sequence of three characters, those characters are ASCII ESC (HEX 1B), ASCII G(HEX 47) and the third character is ASCII H (HEX 48) or M (HEX 4D). Third character represents resolution, there are two possible graphics modes, and those are **high resolution graphic mode** (256 x 192 pixels) represented by sequence <ESC><G><H> and **medium resolution graphic mode** (128 x 96 pixels)which is represented by <ESC><G><M> sequence.

After header sequence, data sequence starts, basic data sequence consists of a pair of run length encoded ASCII characters. The first number represents number of the background (turned off) pixels and the second character is the number of foreground (turned on) pixels. Each number of a pair represents the count number of pixels plus 32 decimal, i.e. from each number 32 is subtracted and that number represents how many next pixels will be turned on or turned off depending on what number of pair we observe. Usually it is used ASCII ~(HEX 7E, DEC 126) as highest possible value, because some terminals interpret ASCII character 7F HEX as <RUBOUT>, because RLE file format was used as file which was to show graphic on terminals. Previous facts lead to conclusion that in each byte we can denote repetition of 94 pixels (126 - 32). For example pair <D><'> (HEX: 44 27, DECIMAL 68 39) means next 68 (decimal) pixels are turned off and then 39 (decimal) pixels are turned on.

Data in file is written in such a way that if the last pixel set was on position 254 then the next pixel will be on the first position in next line i.e. pictures are being drawn from up to down. Let's illustrate this with an example; if the last pixel set on line was on position 252 and data sequence consists of pair , i.e. one background pixel and seven foreground pixels then following pixel is turned off, then the following two pixels of current line are turned on, and then the rest of five pixels turned on, on the beginning of the next line.

The ending sequence for RLE standard consists of three characters <ESC><G><N>, <ESC> is a control character which ends the graphic display. Basic convention is that control character shouldn't affect the display. All control characters should be ignored besides <ESC> and <BEL> characters, <BEL> can be optionally used, so in some cases RLE file ending sequence consists of <BEL><ESC><G><N>. In other words end of RLE file according to standard is <ESC><G><H> or <BEL><ESC><G><N>.

```
1B 47 48 7E 20 7E 20 7E 20 7E 20 ....
.  .  .  .  .  .  .  .  .  .  .  .
.  41 36 .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .  .  .
.  .  .  07 1B 47 4E
```

1B 47 48 - is header <ESC><G><H> and represents high resolution, first data sequence pair 20_{16}, $7E_{16}$ means that first 94_{10} pixels are all turned on, the second data sequence is the same so second 94d pixels are also turned on (the first 188d pixels are turned on so far), and so on. Then somewhere in the file pairs 41h 36h occurs which means that next 33d pixels are turned off and after that 22d pixels are turned off, etc. Last four character are the ending sequence which was described above.

2.2 MS Windows standard for RLE file format

MS Windows standard for RLE have the same file format as well-known BMP file format, but it's RLE format is defined only for 4-bit and 8-bit color images.

Two types of RLE compression is used 4bit RLE and 8bit RLE as expected the first type is used for 4-bit images, second for 8-bit images.

2.3 4bit RLE file format

Compression sequence consists of two bytes; first byte (if not zero) determines number of pixels which will be drawn. The second byte specifies two colors, high-order 4 bits (upper 4 bits) specifies the first color, low-order 4bits specifies the second color this means that after expansion 1st, 3rd and other odd pixels will be in color specified by high-order bits, while even 2nd, 4th and other even pixels will be in color specified by low-order bits. If first byte is zero then the second byte specifies escape code. (See table below)

Second byte	Definition
0	End-of-line
1	End-of-Rle(Bitmap)
2	Following two bytes defines offset in x and y direction (x is right, y is up). The skipped pixels get color zero.
>=3	when expanding following >=3 nibbles (4bits) are just copied from compressed file, file/memory pointer must be on 16bit boundary so adequate number of zeros follows

Table 1. Definition of escape codes (the first byte of compression sequence is 0)

Examples for 4bit RLE:

Compressed data	Expanded data
06 52	5 2 5 2 5 2
08 1B	1 B 1 B 1 B 1 B
00 06 83 14 34	8 3 1 4 3 4
00 02 09 06	Move 9 positions right and 6 up
00 00	End-of -line
04 22	2 2 2 2
00 01	End-of-RLE(Bitmap)

2.4 8bit RLE file format

Sequence when compressing is also formed from 2 bytes, the first byte (if not zero) is a number of consecutive pixels which are in color specified by the second byte.
Same as 4bit RLE if the first byte is zero the second byte defines escape code, escape codes 0, 1, 2, have same meaning as described in Table 1, while if escape code is >=3 then when expanding the following >=3 bytes will be just copied from the compressed file, if escape code is 3 or other greater odd number then zero follows to ensure 16bit boundary.

Examples for 8bit RLE

Compressed data	Expanded data
06 52	52 52 52 52 52 52
08 1B	1B 1B 1B 1B 1B 1B 1B 1B
00 03 83 14 34	83 14 34
00 02 09 06	Move 9 positions right and 6 up
00 00	End-of -line
04 2A	2A 2A 2A 2A
00 01	End-of-RLE(Bitmap)

2.5 Example of RLE usage in other file formats

RLE scheme which will be described in this chapter is being used in PDF and TIFF file format. RLE encoded data consists of compression sequences, one compression sequence starts with number n (byte), this byte may be followed by 1 to 128 bytes, so this 2 to 129 bytes form one compression sequence.

If n is between 0 and 127 inclusive then following n+1 (1 to 128) bytes are just copied during decompression. If n is between 129 and 255 inclusive then the byte which follows n is being copied 256-(n-1) i.e. 2 to 128 times in decompressed file. If 128 occurs then we have reached the end of compressed data.

This scheme is similar to **PackBits** encoding scheme known to Macintosh users.

Examples:

Compressed data-hex format	Decompressed data-hex format
07 A4 56 C9 90 E5 F1 DB 32	A4 56 C9 E5 F1 DB 32
02 23 A1 56	23 A1 56
FE 12	12 12 12
FC 6C	6C 6C 6C 6C 6C

3. Compression and Decompression Algorithm

The RLE algorithm implementation doesn't need data structures like lists or trees. The data structure used not only by RLE but by all compression algorithms is the file: the source file and the destination file are the entry parameters for the compression program and the decompression one.

3.1 Compression algorithm – RLEc.cpp

The source files are read sequentially and are used two variables to identify the streams of characters with same value. The first variable (char OldChar) represents the first character from a run (repeating values is called a *run*), and the second one (char CurrentChar) represents the current character. At the end of a run, if the number of that character repetitions (the variable unsigned char RepetNumber is the repetition counter) is bigger than 3, then we code the text : in the destination file we write the three byte code word shown in Fig 1. In this case the control character is the variable char RepetCode=255, and the variable RepetNumber is smaller than 254.

It is important to realize that the encoding process is effective only if there are sequences of 4 or more repeating characters because three characters are used to conduct RLE so for instance coding two repeating characters would lead to expansion and coding three repeating characters wouldn't cause compression or expansion.

In this way we code the source text by reading it once.

Compression algorithm (and the decompression one) is implemented as a procedure that receives the source file and the destination file.

```
void CompressionRLE (FILE *SourceFile, FILE *CompressedFile)
{     //...variables declaration
        fread(&OldChar,1,1,SourceFile);
        while(fread(&CurrentChar,1,1,SourceFile)==1)
        { if ((RepetNumber<254)&&(CurrentChar==OldChar)) RepetNumber++;
          else
          {  if(RepetNumber>3)
             { fwrite(&RepetCode,1,1,CompressedFile);
               fwrite(&RepetNumber,1,1,CompressedFile);
               fwrite(&OldChar,1,1,CompressedFile);
             }
             else
               for (i=1;i<=RepetNumber;i++)
                 fwrite(&OldChar,1,1,CompressedFile);
               RepetNumber=1;
               OldChar=CurrentChar;
```

```
        }
      }
      if(RepetNumber>3)
      {  fwrite(&RepetCode,1,1,CompressedFile);
         fwrite(&RepetNumber,1,1,CompressedFile);
         fwrite(&OldChar,1,1,CompressedFile);
      }
      else
        for(i=1;i<=RepetNumber;i++) fwrite(&OldChar,1,1,CompressedFile);
}
```

The main program (RLEc.exe) receives the name of the source file (and optionally the name of the destination file). The source file extension is kept in the destination file because we want decompression process to have as result the original file.

The RLE compression process is without any information losses. At the end of the main program we call a procedure from the library file „lib.h" that calculates the length of the two files and the compression ratio with the formula :

CompressionRatio=100 – (DestinationLenght*100/SourceLenght)

3.2 Decompression Algorithm – RLEd.cpp

```
void DecompressionRLE(FILE *SourceFile,FILE *CompressedFile)
{  char RepetCode=225;
   unsigned char i,RepetNumber=1;
   char RepetChar,CurrentChar;
   while(fread(&CurrentChar,1,1,SourceFile)==1)
   {  if(CurrentChar==RepetCode)
      {  fread(&RepetNumber,1,1,SourceFile);
         fread(&RepetChar,1,1,SourceFile);
         for(i=1;i<=RepetNumber;i++) fwrite(&RepetChar,1,1,CompressedFile);}
      else fwrite(&CurrentChar,1,1,CompressedFile); } }
```

The test files results :

Source file	Before compression (bytes)	After compression (bytes)	Compression ratio (%)
Test01.doc	37376	24454	35
Test02.jpg	25959	25469	2
Test03.exe	47061	47052	1
Test04.dat	2976	2122	29
Test05.txt	3050	3054	0
Test06.htm	11049	10511	5
Test07.ani	12144	5770	53
Test08.obj	2875	2790	3
Test09.gif	2935	2928	1
Test10.z80	45930	45787	1
Test11.pod	3001	2156	29

4. Conclusions

- RLE is usually applied to the files that a contain large number of consecutive occurrences of the same byte pattern.
- RLE may be used on any kind of data regardless of its content, but data that is being compressed by RLE determines how good compression ratio will be achieved. So RLE is used on text files which contains multiple spaces for indention and formatting paragraphs, tables and charts. Digitised signals also consist of unchanged streams so such signals can also be compressed by RLE. Good examples of such signal are monochrome images, and questionable compression would be probably achieved if such compression was used on continuous-tone (photographic) images.
- Fair compression ratio may be achieved if RLE is applied on computer generated colour images

For a monochrome image, the alphabet contains two symbols. The elements in a monochrome image can be : 1 for white pixel, and 0 for black pixel.

For example the stream:

00000000001111111111000001111

can be compress like this :

10 8 5 4

For this type of files we use a modified version of the algorithm. In the destination file we will have:

- **the pixel that starts the sequence**;
- **repetition number of that pixel;**
- **repetition number for each color.**

If we have a sequence of more than 255 characters with the same value, it will be coded like this:

- **255**;
- **0**;
- **repetition number – 255**.

For example, the stream:

……0000…….0000011111…….

(we have a sequence of 300 characters with value = 0)
become after compression:

…….255 0 45 3……

- RLE is a loss-less type of compression and cannot achieve great compression ratios, but a good point of that compression is that it can be easily implemented and quickly executed.
- We have a maximum ratio when the text to be compressed is very big and it has only one character that is repeating.

We have a ratio equals with 0 when the text with L symbols has L different characters or L/3 runs of different characters.

Bibliography

1. **Pagina de internet :**
   www.rasip.fer.hr/research/compress/algorithms/fund/rl

2. **[Ivan98] Ion Ivan, Daniel Verniş – Compresia de date, Editura CISON, Bucureşti 1998.**