New Algorithms For Random Access Text Compression

Robert. L
Dept. of Computer Science
Government Arts College
Coimbatore, INDIA

Nadarajan. R
Dept.of Mathematics and Computer Applns.
PSG College of Technology
Coimbatore, INDIA

Abstract

Random Access text compression is a type of compression technique in which there is a direct access to the compressed data. It facilitates to start decompression from any place in the compressed file, not necessarily from first. If any byte changed during transmission, the remaining data can be retrieved safely. In this paper a try has been made to develop few algorithms for Random access text compression based on the Byte Pair Encoding Scheme[1]. The BPE algorithm relies on the fact that ASCII character set uses only codes from 0 through 127. That frees up codes from 128 through 255 for use as pair codes. Pair code is a byte, used to replace the most frequently appearing pair of bytes in the text file. Five algorithms are developed based on this Byte Pair Encoding Scheme. These algorithms finds the unused bytes at each level and tries to use those bytes for replacing the most frequently used bytes.

1. Introduction

Virtually all modern compression methods are adaptive models and generate variable-bit-length codes that must be decoded sequentially from beginning to end. The Huffman algorithm[4] uses the notion of variable length prefix code for replacing characters of the text. The code depends on the input text , and more precisely on the frequencies of characters in the text. The most frequent characters are given shortest code words, while the least frequent symbols correspond to the longest code words. The basic idea of Arithmetic Coding[7] is to consider symbol as digits of a numeration system, and texts as decimal parts of numbers between 0 and 1. The length of the interval attributed to a digit is made proportional to the frequency of the digit in the text. Ziv and Lempel[6] designed a compression method using encoding segments. These segments of the original text are stored in a dictionary that is built during the compression process. In this model where portions of the text are replaced by pointers on previous occurrences, the Ziv-Lempel compression can be proved to be asymptotically optimal. Michael Burrows and David Wheeler[3] recently released the details of a transformation function that opens the door to some revolutionary new compression techniques. The Burrows-Wheeler Transformation, transforms a block of data into a format that is extremely well suited for compression. It takes a block of data and rearranges it using a sorting algorithm. The resulting output block contains exactly the same data elements that it started with, differing only in their ordering. The transformation is reversible. In these models, random access to large data sets can be provided only by dividing the data into smaller files or blocks and compressing each chunk separately. Even if an application is sophisticated enough to decompress separate blocks of data, it still must begin decompressing each block at the start of that block. There is a simple text compression trick that allows random access; it employs the unused high order bit in ASCII characters to indicate that the preceding space character has been removed. This technique in effect removes all single spaces and reduces runs of consecutive spaces to half length, compressing typical text by 15 percent[2]. Another simple approach is to encode common words as one byte using the fixed dictionary of the most frequently used words. But schemes such as these are not general purpose and have limited usefulness. Byte Pair Encoding scheme[1] is a universal compression algorithm that supports random access for all types of data. The global substitution process of BPE produces a uniform data format that allows decompression to begin anywhere in the data. Using BPE, data from anywhere in a compressed block can be immediately decompressed without having to start at the beginning of the block. This can provide a very fast access for some applications. It is like being able to grab a volume of an encyclopedia off the shelf and open it to the exact page you want without having to flip through all the earlier pages in the volume.

1.1. BPE Algorithm

ASCII character set uses only bytes from 0 to 127. This method uses 128 to 255 for replacing the most frequently occurring pairs of adjacent bytes. The BPE operates by finding the most frequently occurring pair of adjacent bytes in the text and replacing all instances of the pair with a code from 128 to 255. The compressor repeats this process until no more compression is possible or all codes from 128 to 255 are used for replacement. The algorithm records the details of the replacements in a table, which is stored as a header in the compressed file. The Process of compression is illustrated by the following

Proceedings of the Third International Conference on Information Technology: New Generations (ITNG'06)
0-7695-2497-4/06 $20.00 © 2006 IEEE

example. Consider the following line of text and their ASCII values.

A B A B C A B C D C D A B C D B C D

65 66 65 66 67 65 66 67 68 67 68 65 66 67 68 66 67 68

Replace the most frequently occurring pair 66 67 by the code 128

65 66 65 128 65 128 68 67 68 65 128 68 128 68

Replace the pair 65 128 by the code 129

65 66 129 129 68 67 68 129 68 128 68

The Compressed file will be

130 66 67 65 128 65 66 129 129 68 67 68 129 68 128 68

Input bytes = 18 Output bytes = 16

In this compressed file the first byte represents the last pair code used followed by pair table and compressed text. This algorithm uses a stack type data structure for expanding the pair codes into pairs. The algorithm first reads the pair table from the input stream, and stores it in a buffer, then processes each byte of compressed data, maintaining a small-stack of partially expanded bytes. The next byte to be processed is popped from the stack, or if the stack is empty, read from the input file. If the byte is a pair code, the pair of bytes is looked up in the table and pushed onto the stack; otherwise the byte is output as a normal character.

2. Design of Random Access Compression algorithms

In this paper some variances of BPE algorithm and their implementation are introduced and discussed.

2.1. Algorithm-1

ASCII character set uses only bytes from 0 to 127. This method uses 128 to 255 for replacing the most frequently occurring pairs of adjacent bytes. The main key of BPE algorithm is, it uses the bytes 128 to 255 for replacing the most frequently occurring pair of bytes. All the text files are not using all the bytes from 0 to 127. Particularly the control characters from 0 to 31, except blank, end of line bytes, are very rarely used. So, this improved version of BPE, uses those unused bytes along with 128 to 255, for replacing the most frequently occurring pair of bytes. Two tables are maintained here for recording information about replacements. One table of maximum size 128x2, which keeps the pairs that are replaced by codes 128 to 255. The second table is for keeping the unused bytes and their corresponding pairs that are replaced. Let us name it as UNUSED table. It is an Nx3 array, where N is the number of unused bytes.

2.1.1. Compression. The algorithm starts with finding all unused bytes from 0 to 127 and stores them in the 0th column of the UNUSED table. Then, it finds the most frequently occurring pairs and replaces with code 128 through 255, and store the information about these replacements in the pair table. Then, the algorithm replaces the most frequently occurring pair, if any, by the unused bytes that are stored UNUSED table. It records the replacements by the unused byte in the table. This process continues until there is no pair for replacement or all the unused bytes are used for replacements. The following algorithm illustrates this compression process.

```
1       Read file into buffer
2       Find the unused bytes from 0 to127 and store
them in UNUSED[i][0],for i varies from 0 to nb-l, where
nb is number of unused bytes.
3       Construct the Pair table and replace the most
frequently occurring pair
4       Write the Pair table in to the output file
5       index = 0
6       While ( Index < nb )
        {
7               Code = UNUSED [index] [0]
8               Find the most frequently occurring pair
9               If there is no such pair, Break
10              Add the pair to UNUSED[Index][1],
                                UNUSED[Index][2]
11              Replace all such pairs by Code
12              Index++
        }
13      Write Index to the output file
14      Write UNUSED table to the output file
15      Write buffer to the output file
```

2.1.2. Decompression. In this method, the decompression is performed in two stages. The compressed code is decompressed by using the table UNUSED in the first stage. The algorithm finds the code stored in UNUSED[i][0] and replaces with the pair stored in UNUSED[i][1] and UNUSED[i][2]. In the second stage, the resulted file from the first stage again decompressed using the pair table. That is, the algorithm finds all the codes from 128 to 255 from the given file, and replaces those code by the pair stored in PAIR[code-128][0] and PAIR[code-128][1]. The following algorithm illustrates this process.

```
1       Read and Store PAIR table into the Buffer
2       Read and Store UNUSED table into the Buffer
3       Initialise Stack
4       While (Stack not Empty OR Not EOF(input
file))
        {
5               If Stack Empty, read a Byte from input file
                 Else Pop Byte from Stack
6               If Byte is in UNUSED[i]0] for any i
                                from 0 to np-l then
7                   Push the pair UNUSED[i][2],
                                UNUSED[i][1] into stack
                Else Write Byte on output file FILE1
        }
8       While (Stack not Empty OR Not EOF(FILE1))
        {
9               If Stack Empty, read Byte from FILE1
                else Pop Byte from Stack
```

```
10              If Byte is in PAIR table then
                Push the pair into stack
                else Write Byte on output file FILE2
        }
```

2.2. Algorithm-2

The algorithm discussed in section 2.1 uses the bytes from 128 to 255, and unused bytes from 0 to 127, for replacing the most frequently appeared pair of bytes. Because of the successive replacements, after the compression, we find, some of the bytes that are used in the file are become unused after the compression. For example.

A	B	A	B	C	D	B
65	66	65	66	67	68	66
128	128	67	68	66		

Here the byte 65 becomes unused after the replacements. So, by using those bytes as codes, again apply the process of finding the most frequently occurring pair. This process is repeated until all possible bytes are there in the resulted file or there is no frequently occurring pair.

2.2.1. Compression. In this algorithm, compression process involves many stages. At the first stage of the compression, the algorithm finds the most frequently occurring pair and is replaced by the bytes from 128 to 255. In the second Stage, it finds all the unused bytes in the resulted file of stage-1, and replaces the most frequently occurring pairs by those unused bytes. At the third stage, the algorithm finds the unused bytes after stage-2, and replaces the most frequently occurring pair by those unused byte. Like this, the process is continued until, either there is no frequently occurring pair or all possible bytes are used in the resulting file from the previous stage. At each stage, the replacements are recorded and written into the output file. The number of stages is written as a first byte in the compressed file. The following algorithm illustrates this process.

```
1       Read file into the Buffer
2       Construct the PAIR table and replace the most-
        frequently occurring pair as per BPE
3       Write the PAIR table into the Output file
4       pair_over = false
5       do
        {
6           Find the unused bytes, if any, and store
            into UNUSED[i][0], for i from 0 to nb-l,
7           If (nb= = 0 ) then break
8           Index = 0
9           While( Index<nb )
            {
10              Code=UNUSED[index][0]
11              Find the most frequently appeared pair
12              If there is no such pair Then
                        { pair_over = true; break}
13              Add pair to the UNUSED[Index][1] and
                            UNUSED[Index][2]
14              Replace all such pairs by Code
15              Index++
            }
16          Write Index into the Output file
17          Write UNUSED table into the Output file
18          If (pair_over) Then break
19      }While(True)
20      Write the Buffer into the output file.
```

2.2.2. Decompression. This algorithm just performs the reverse process of compression explained in section 2.2.1. The decompression is performed on multi stages as like compression. If n stages are performed at compression, then decompression also involves n stages. Firstly, the algorithm starts by reading the n UNUSED tables, and a PAIR table, from the compressed file. The algorithm, then process the nth stage of UNUSED table, by finding the code which are in UNUSED[code][0], and replacing that by pair UNUSED[code][1],UNUSED[code][2] in the compressed data. The algorithm does the same process for n-l, n-2,…, 1 stages. Here, at each stage the input file is the output file of the previous stage. At last, the algorithm finds for the byte from 128 to 255, and is replaced by the pair from the PAIR table. The following algorithm illustrates this process

```
1   Read and Store Pair table in a Buffer
2   Read number of stages-(Assume n) from input file
3   Read the set of n UNUSED tables and
        Store it in a 3D array named as Stage
        Stage[O][ ][ ] ← Oth stage table
        Stage[ 1 ][ ][ ] ← 1st stage table
        :
        Stage[n-1][][] ← (n-1) th stage table
4   st=n-1
5   Copy the input file to the TEMP-FILE1
6   Initialise Stack
7   do
    {
8    While( Stack not Empty OR not EOF(TEMP-FILE1))
       {
9          If Stack is Empty, Read byte from FILE1
             Else Pop byte from stack
10         If byte is in Stage[st][i][O] for any i Then
             Push the pair Stage[st][i][2], Stage[st][i][l]
             Else Write Byte into file TEMP-FILE2
       }
11     Copy  from TEMP-FILE2 to TEMP-FILE1.
12     st--
13  } While ( st >= 0 )
14  Initialise Stack
15 While(Stack Not Empty OR Not EOF(TEMP-FILE1))
    {
16     If Stack is Empty, Read byte from FILE1
           Else Pop byte from stack
17     If byte is in PAIR table Then
                Push the pair into Stack
           Else Write Byte on Output File
       }
```

2.3. Algorithm-3

In this version of the algorithm, a try is made to replace most frequently occurring three bytes(tribytes), by

the code from 128 to 255. The idea for replacing tribytes, is evolved from the fact that the most of the English words have prefixes, suffixes with length 3. Example: ing, ted, sed, ies, dis, etc. While replacing, if there are no more adjacent tribytes, then the algorithm replaces the most frequently occurring pairs. The details of replacements are kept in two tables. One table used to keep the replacements of tribytes. The details of the replacement of pairs are recorded in the PAIR table. It is necessary to store the number of entries in the TRI table and PAIR table in the compressed file other than the actual tables itself.

2.3.1. Compression. The algorithm starts by replacing the most frequently occurring tribytes by the codes 128 through 255. If there are no more tribytes, and still there exist some bytes from 128-255 to use, the algorithm starts finding and replacing most frequently appeared pairs. The following algorithm illustrates this process. This algorithm can be further improved by using the unused bytes also as code for replacing Tris and Pairs.

```
1       Read File into the buffer
2       Code = 128
3       do
        {
4          Find a most frequent Tribyte in the Buffer
5          If  No such TriBytes, Then Break
6          Add TriBytes into a TRI table
7          Replace all such TriBytes by Code
8          Code++
9       } While( Code < = 255)
10      Write Code into the Output file
11      Write TRI table into the Output file
12      While (Code<=255)
        {
13         Find the most frequent byte pair in buffer
14         If  No such pair Then Break
15         Add pair to a PAIR table
16         Replace all such pair with the code
17         Code ++
        }
18      Write Code into the Output file
19      Write PAIR table into the Output file
20      Write buffer into the Output file
```

2.3.2. Decompression. In this method, the decompression process consists of two stages. Replacing pair codes by pair of bytes, and replacing tricodes by tribytes. The following algorithm illustrates this process.

```
1       Code = the first Byte of the Inputfile
2       Read the TRI table and Store in a Buffer
3       Initialise Stack
4       If ( Code!=255 ) then
        {
5          Read and Store the PAIR table in a Buffer
6          While (Stack not Empty OR Not EOF(input) )
           {
7             If Stack empty, Read byte from input file
                    Else Pop Byte from Stack
8             If Byte is a Pair code then
                    Push the pair onto stack
                    Else Write the Byte to the file FILE1
           }
        }
        else Copy the Input file to file FILE1
9       Initialise Stack
10      While (Stack not Empty OR Not EOF(FILE1))
        {
11         If Stack empty, Read byte from FILE1
                    Else Pop Byte from Stack
12         If Byte is a TRI code  then
                    Push the Tri bytes onto stack
                    Else Write the Byte to the output file
        }
```

2.4. Algorithm-4

Many English words are having the length four or having prefixes and suffixes with length four like sion, tion,,,etc. So, In a text file, there may be a repetition of four consecutive bytes many times. This algorithm starts finding those most frequently occurring four bytes, and are replaced by the code from 128 to 255. After replacing, if any code from 128 to 255 remains unused, the algorithm starts finding the frequent adjacent tri bytes and are replaced. After, still if there are bytes from 128 to 255 that are unused, that is if number of frequent quads and tris replaced are less than 128, then the algorithm starts replacing the frequently appeared pair by remaining codes. This algorithm significantly improves the compression ratio. This algorithm uses 3 tables. The PAIR table, and TRI table are used for storing the replaced pair of bytes and tri bytes. Other than these tables, another table QUAD is used to store the most frequently appeared quad of bytes.

2.4.1. Compression. In this method, the Compression starts by replacing Quads, then Tris, and then pairs by the codes from 128 through 255.

```
1       Read File into the buffer
2       Code = 128
3       do
        {
4          Find the most frequent quad in Buffer
5          If  No such quads, Then Break
6          Add quad into a QUAD table
7          Replace all such quads by Code
8          Code++
9       } While( Code <= 255)
10      Write Code into the Output file
11      Write QUAD table into the Output file
        While( code<=255)
12      {
13            Find the frequent Tribyte in Buffer
14            If No such TriBytes, Then Break
15            Add TriByte into a TRI table
16            Replace all such TriBytes by Code
17            Code++
        }
18      Write Code into the Output file
19      Write TRI table into the Output file
20      While (Code<=255)
        {
21            Find the frequent byte pair in buffer
22            If No such pair Then Break
23            Add pair to a PAIR table
```

```
24                  Replace all such pair with the code
25                  Code ++
            }
26          Write Code into the Output file
27          Write PAIR table into the Output file
28          Write buffer into the Output file
```

2.4.2. Decompression. The decompression process consists of three stages. Replacing pair codes by pair of bytes using PAIR table, and replacing tri code by tri bytes using TRI table, and replacing quad codes by quad of bytes using QUAD table. The following algorithm illustrates this process.

```
1  Read the first Byte of Input file and assign to QuadN
2  Read the QUAD table and Store in a Buffer
3  Intialise Stack
4  If( QuadN!=255) then
     {
5           Read a Byte and assign to TriN
6           Read and Store the TRI table in a Buffer
7           if (TriN !=255)
            {
8            Read a Byte and assign to PairN
9            Read and Store the PAIR table in a Buffer
10           While (Stack not Empty OR Not EOF(input) )
            {
11            If Stack empty, Read byte from input file
                    Else Pop Byte from Stack
12            If Byte is a Pair code then
                    Push the pair onto stack
                    Else Write the Byte to the output file
            }
13          copy the output file to the inputfile
            }
14          Initialise Stack
15          While (Stack not Empty OR Not EOF(input) )
            {
16            If Stack empty, Read byte from input file
                    else Pop Byte from Stack
17                  If Byte is a TRI code then
                    Push the Tri bytes onto stack
                    else Write the Byte to the output file
            }
18          Copy the output file to the inputfile
            }
19          Initialise Stack
20          While (Stack not Empty OR Not EOF(input"
            {
21            If Stack empty, Read byte from input
                    else Pop Byte from Stack
22           If Byte is a QUAD code then
                    Push the quad bytes onto stack
                    else Write the Byte to the output file
            }
```

2.5. Algorithm-5

This algorithm simply combines the concept of Algorithm 2 and Algorithm 4. Algorithm 4, uses only the bytes from 128 to 255. This algorithm starts with algorithm 4, for the given file and the output of the algorithm 4, is processed further. It finds all unused bytes in the resulted file and finds all frequent pairs and are replaced by those unused bytes. This process is repeated until all bytes are used in the resulted file or no more pairs to replace. This algorithm gives better compression ratio compare with all the algorithms discussed.

2.5.1. Compression. In this method, the input file to be compressed has to be given to the Algorithm-4 discussed in section 2.4, and the resulted file is given as input to the following algorithm.

```
1    Read the resulted file into the buffer
2    do
     {
3       Collect all unused bytes from the buffer
4        For each unused byte
         {
5        Find the most frequently appeared pair
6        Replace the  pair by a unused byte
7        Store the details of the replacement into UNUSED
         }
8        Write UNUSED table into the output file
9     } While(if there any unused Byte in the Buffer)
10   Write the Buffer into the output file.
```

2.5.2. Decompression. The decompression is done in two major stages. In the first major stage the compressed file is decompressed by using the following algorithm, which replaces the pair codes by the pair of bytes by using the UNUSED tables:

```
1    Read number of stages from input file, assign to n.
2    Read the set of n UNUSED tables and
     Store it in a 3'D array named as Stage
3    st=n-l
4    Copy the input file to the file FILE1
5    Initialise Stack
6    do
      {
7       While( Stack not Empty OR not EOF(FILE1))
        {
8          If Stack is Empty, Read byte from FILE1
                  Else Pop byte from stack
9          If byte is in STAGE[st][i][O] Then Push the pair
                  STAGE[st][i][2],STAGE[st][i][1]
                  Else Write Byte into file FILE2
        }

10      copy the file from FILE2 to FILE 1. .
11      st = st -1
12    } While (st >= 0)
```

In the second stage the FILE1 is given as input to the decompression Algorithm explained in section 2.4.2. The outcome of this, is a decompressed file.

2.6. Experimental Results

The Table-1 contains a sample experimental results showing the behaviour of the BPE and developed algorithms on different types of texts. The algorithms was run on a machine with 433Mhz processor speed, 128K of Cache, and 640K of main memory. In these methods the decompression time taken by all the algorithms on these source file is less than a single hundredth of a second.

The source files are
File1 - DOS 6.22 Help File -Networks.txt
File2 - C Source file
File3 - A file containing a repetition of abc...zABC...Z
File4 - Random text file generated using rand( ) function

Table 1. Compression Ratio

Source Texts Sizes in Bytes	File1 17465	File2 8331	File3 10400	File4 8120
BPE	9116 48%	3778 55%	119 99%	7906 3%
Algorithm-1	8350 53%	3413 60%	119 99%	7907 3%
Algorithm-2	8094 54%	2874 66%	120 99%	7908 3%
Algorithm-3	9333 47%	3499 59%	102 100%	7911 3%
Algorithm-4	9529 46%	3482 59%	88 100%	8012 2%
Algorithm-5	7857 56%	2865 66%	89 100%	7914 3%

3. Conclusion

These algorithms compresses typical text files approximately half of their original size, but of course the actual amount of compression depends on the data being compressed. The behavior of the compression algorithms are exhibited through the experimental table. Considering the time taken for compression, most of the time is spent on searching for the most frequently occurring pairs, Tris and quads. But decompression is very fast in all these algorithms. The third and fourth algorithms can be improved by combining the concept of algorithm 2 as like algorithm 5. Another major issue in all these algorithms is, deciding the threshold value for replacement. That is, replacement of pairs, tris, quads can be done only when the number of occurrences of pair, tri or quad, should exceed the threshold value. So, based on the threshold value, the compression ratio may differ. An algorithm may be designed for finding the suitable threshold value, which increases the compression ratio for the given text file. Some more work necessary to improve the searching process on these algorithms, which reduces the compression time.

References

[1] Philip Gage, "Random Access Data Compression", The C/C++ Users Journal, Sep 1997.

[2] Robert.L., Nadarajan.R., "New Algorithms for Random Access Text Compression", M.Phil.Thesis, Bharthiar University, INDIA, Dec 1999.

[3] M.Burrows and D.Wheeler, "A Block sorting lossless data compression algorithms", Technical report 124, Digital Equipment Corporation, Palo Alto, California, 1994

[4] Huffman. D.A., "A method for the construction of minimum redundancy codes", Proceedings of the I.R.E. 40, 1098-1101, 1951.

[5] J.Cleary and J. Witten, "Data Compression using adaptive coding and partial string matching", IEEE Transactions on Communications, pp 396-402, 1984

[6]J.Ziv and A.Lempel, "A universal algorithm for sequential Data Compression", IEEE transaction on information theory, pp. 337-342, 1977.

[7] Witten I.H. , Neal.R., and Cleary J., "Aritmetic coding for Data Compression", Comm. ACM, 30(6), 520-540, 1987.

[8] Gallager R.G., "Variations on a theme by Huffman", IEEE trans. Inf. Theory, 24(6), 668-674, 1978.

[9] Cormack G.V. and Horspool R.N.S., "Algorithms for adaptive Huffman codes", Inf. Process Lett. 18(3), 159-165, 1984.

[10] Welch T.A., "A technique for high performance Data Compression", IEEE Computer, 17(6), 8-19, 1984.

[11] Maxime Crochemore and Thierry Lecroq, "Text Data Compression Algorithms", CRC press, 12, 1-22, 1999.

[12] David Salomon, Data Compression-The Complete Reference, Springer, New York, 2000.