A New Algorithm for Data Compression

Philip Gage

1994

Phil Gage is a software engineer in Colorado Springs and has a BS degree in computer science from the University of Colorado. He has been a professional programmer since 1983 and has used C since 1986. Phil can be reached at (719) 593-1801 or via CompuServe as 70541,3645.

Abstract

Data compression is becoming increasingly important as a way to stretch disk space and speed up data transfers.

This article describes a simple general-purpose data compression algorithm, called Byte Pair Encoding (BPE), which provides almost as much compression as the popular Lempel, Ziv, and Welch (LZW) method [3, 2]. (I mention the LZW method in particular because it delivers good overall performance and is widely used.) BPE's compression speed is somewhat slower than LZW's, but BPE's expansion is faster. The main advantage of BPE is the small, fast expansion routine, ideal for applications with limited memory. The accompanying C code provides an efficient implementation of the algorithm.

1 Theory

Many compression algorithms replace frequently occurring bit patterns with shorter representations. The simple approach I present is to replace common pairs of bytes by single bytes.

The algorithm compresses data by finding the most frequently occurring pairs of adjacent bytes in the data and replacing all instances of the pair with a byte that was not in the original data. The algorithm repeats this process until no further compression is possible, either because there are no more frequently occurring pairs or there are no more unused bytes to represent pairs. The algorithm writes out the table of pair substitutions before the packed data.

This algorithm is fundamentally multi-pass and requires that all the data be stored in memory. This requirement causes two potential problems: the

Originally appeared in The C User Journal under DOI 10.5555/177910.177914. Reconstructed from http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM to PDF in 2020 by Leon Derczynski.

algorithm cannot handle streams, and some files may be too large to fit in memory. Also, large binary files might contain no unused characters to represent pair substitutions.

Buffering small blocks of data and compressing each block separately solves these problems. The algorithm reads and stores data until the buffer is full or only a minimum number of unused characters remain. The algorithm then compresses the buffer and outputs the compressed buffer along with its pair table. Using a different pair table for each data block also provides local adaptation to varying data and improves overall compression.

Listing 1 and Listing 2 show pseudocode for the compression and expansion algorithms.

2 Implementation

Listing 3 and Listing 4 provide complete C programs for compression and expansion of files. The code is not machine dependent and should work with any ANSI C compiler. For simplicity, error handling is minimal. You may want to add checks for hash table overflow, expand stack overflow and input/output errors. The expansion program is much simpler and faster than the compression program.

The compression algorithm spends the most time finding the most frequent pair of adjacent characters in the data. The program maintains a hash table consisting of arrays *left[]*, *right[]*, and *count[]* to count pair frequencies. The hash table size *HASHSIZE* must be a power of two, and should not be too much smaller than the buffer size *BLOCKSIZE* or overflow may occur. Programmers can adjust the value of *BLOCKSIZE* for optimum performance, up to a maximum of 32767 bytes. The parameter *THRESHOLD*, which specifies the minimum occurrence count of pairs to be compressed, can also be adjusted.

Once the algorithm finds the most frequently occurring pair, it must replace the pair throughout the data buffer with an unused character. The algorithm performs this replacement in place within a single buffer. As it replaces each pair, the algorithm updates the hash table's pair counts. This method of updating the hash table is faster than rebuilding the entire hash table after each pair substitution.

3 Pair Table Compression

After the program has compressed a buffer, the pair table contains entries of those pairs of bytes that were replaced by single bytes within the buffer. Figure 1 shows a sample pair table resulting from compression of a string of 9 characters, with a hypothetical character set limited to 8 characters. The pair table does not store the replacement bytes; rather, a pair's position in the table indicates the value of the replacement byte. For example, in Figure 1, pair 'A':'B' is found in the pair table's 8th entry, which indicates that this particular pair was replaced

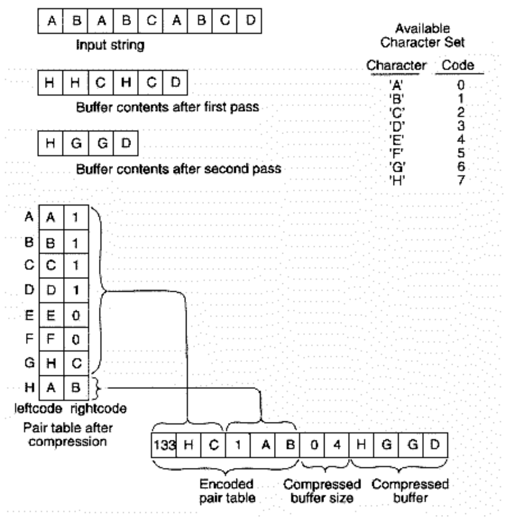

Figure 1: Illustration of compression process with hypothetical character set

by the character 'H'. Those entries in the pair table not containing a replaced pair are distinguished by a left code whose value is equal to its index (*index == leftcodef[index]*). (Note: The compression algorithm uses the array *rightcode[]* for two different purposes. During the initial part of the compression process, function *fileread* uses the *rightcode[]* array to flag used vs. unused characters. After buffer compression, *rightcode[]* serves as half of the pair table.)

The algorithm must write the pair substitution tables to the output along with the packed data. It would be simple just to write the character code and pair for each substitution. Unfortunately, this method would require three bytes per code and would waste space. Therefore, this program applies a form of encoding to also compress the pair table before it is written to the output.

To compress the pair table, the program steps through the table from the first entry thru its last entry, classifying each entry as representing a replaced pair (*index != leftcode[index]*) or as not representing a replaced pair (*index == leftcode[index]*). To encode a group of contiguous replaced pairs, the program emits a positive count byte followed by the pairs. To encode a group of contiguous table entries that don't represent replaced pairs, the program emits a negative count byte followed by one pair.

In the encoded pair table a positive count byte indicates to the expansion program how many of the following pairs of bytes to read, while a negative byte causes the expansion program to skip a range of the character set and then read a single pair. This technique allows many pairs to be stored with only two bytes per code.

To further increase pair table compression, I've modified the algorithm from the preceding description to avoid disrupting runs of pairs where possible. Specifically, the algorithm does not encode an isolated, single byte not rep-

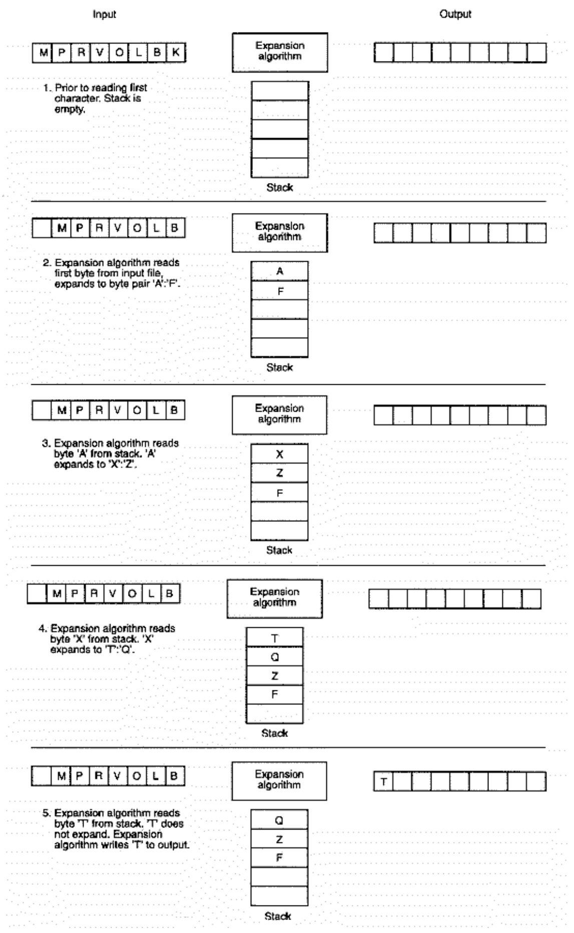
Input
Output
M P R V O L B K
Expansion algorithm
1. Prior to reading first character. Stack is empty.
Stack
2. Expansion algorithm reads first byte from input file, expands to byte pair 'A':'F'.
A
F
3. Expansion algorithm reads byte 'A' from stack. 'A' expands to 'X':'Z'.
X
Z
4. Expansion algorithm reads byte 'X' from stack. 'X' expands to 'T':'Q'.
T
Q
5. Expansion algorithm reads byte 'T' from stack. 'T' does not expand. Expansion algorithm writes 'T' to output.

Figure 2: Illustration of expansion process

resenting a pair; instead, the algorithm writes the byte to output along with the pair data without an accompanying right code. The expansion algorithm knows that the byte does not represent pair data because the byte occurs at a position such that *byte value == leftcode[byte value].*

4 Expansion

As opposed to the compression algorithm, which makes multiple passes over the data, the expansion algorithm operates in a single pass. You can think of the expansion algorithm as a black box which obtains input bytes from one of two sources, the input file, or a stack (see Figure 2) . Regardless of an input byte's source, the algorithm processes each byte according to the following rule: if the byte is a literal, the algorithm passes it to the output; if the byte represents a pair, the algorithm replaces it with a pair and pushes the pair onto the stack.

Now, to complete the loop, the algorithm selects its input source according to the following rule: If the stack contains data, the algorithm obtains its next input byte from the stack. If the stack is empty, the algorithm obtains its next input byte from the input file.

The effect of these rules is "local" expansion of byte pairs; that is, if a byte expands to a pair, and that pair contains one or more bytes in need of expansion, the algorithm will expand the newly created bytes before it reads any more from the input file.

5 Advantages of BPE

One significant advantage of the BPE algorithm is that compression never increases the data size. This guarantee makes BPE suitable for real-time applications where the type of data to be compressed may be unknown. If no compression can be performed, BPE passes the data through unchanged except for the addition of a few header bytes to each block of data. Some algorithms, including LZW, can greatly *inflate* the size of certain data sets, such as randomized data or pre-compressed files.

LZW compression adapts linearly to frequently occurring patterns, building up strings one character at a time. The BPE algorithm adapts exponentially to patterns, since both bytes in a pair can represent previously defined pair codes. The previously defined pair codes can themselves contain nested codes and can expand into long strings. This difference between LZW and BPE provides better compression for BPE in some cases. For example, under BPE a run of 1024 identical bytes in a row is reduced to a single byte after only ten pair substitutions. This nesting of pair codes is the real power of the algorithm. The following example illustrates this process:

```
Original input data string:   ABABCABCD
Change pair AB to unused X:   XXCXCD
Change pair XC to unused Y:   XYYD
```

Finally, both BPE's compression and expansion algorithms require little memory for data arrays, 5 to 30K for compression and only 550 bytes for expansion. The expansion routine is so simple that, coded in assembler, it should require only about 2K of memory for all code and data.

6 Results

The BPE program delivers performance comparable to LZW, as shown in Table 1. I compressed and expanded what I consider to be a typical binary file, the Windows 3.1 program WIN386.EXE. I measured the timing on a 33MHz 486DX PC compatible using MS-DOS 5.0 and Borland C++ 3.0.

	12 bit LZW	14 bit LZW	Default BPE	Small BPE	Fast BPE
Original file size (bytes)	544,789	544,789	544,789	544,789	544,789
Compressed file size (bytes)	299,118	292,588	276,955	293,520	295,729
Compression time (secs)	28	28	55	41	30
Expansion time (secs)	27	25	20	21	19
Compression data size (bytes)	25,100	90,200	17,800	4,400	17,800
Expansion data size (bytes)	20,000	72,200	550	550	550

Table 1: Comparison of LZW and BPE performance

I tested the BPE program against the LZW program, *LZW15V.C*, from *The Data Compression Book* by Mark Nelson [1], using 12-bit codes with a 5021 entry hash table and 14-bit codes with a 18041 entry hash table. The 12-bit version uses less memory for data but does not compress quite as well. I also tested several other LZW programs and obtained similar results.

The Default BPE column shows the results of using the default parameters from Listing 3, which are tuned for good performance on all types of files, including binary and text. Although BPE packed this binary file slightly better than LZW, performance will vary on other files depending on the type of data.

The Small BPE column shows the results of reducing the amount of memory available for the compression program data arrays. I changed *BLOCKSIZE* from 5000 to 800 and *HASHSIZE* from 4096 to 1024. These changes only slightly decreased the compression ratio on the binary file, but the smaller buffer size will not work as well on text files.

The Fast BPE column shows the results of increasing compression speed, by changing *THRESHOLD* from 3 to 10. This change caused the program to skip pairs with less than 10 occurrences. Since the program compresses most frequently occurring pairs first, skipping low-frequency pairs near the end of block processing has little effect on the amount of compression but can significantly improve speed. This change reduced the compression time from 55 to 30 seconds.

7 Enhancing BPE

The BPE algorithm could be enhanced by block size optimization. The block size is critical to both the compression ratio and speed. Large blocks work better for text, small blocks work better for binary data.

8 Conclusion

It's surprising that the BPE algorithm works as well as it does, considering that it discards all information on previous data and does not use variable-sized bit codes, contrary to many modern compression techniques. The BPE compression algorithm is useful for applications requiring simple, fast expansion of compressed data, such as self-extracting programs, image display, communication links and embedded systems. Advantages include a small expansion routine, low memory usage, tunable performance, and good performance on worst-case data. The disadvantages of BPE are slow compression speed and a lower compression ratio than provided by some of the commonly used algorithms, such as LZW. Even with these disadvantages, BPE is a worthwhile technique to have at your disposal.

References

[1] Nelson, M. *The Data Compression Book.* M&M books, 1991.

[2] Welch, T. A. A technique for high-performance data compression. *Computer*, 6 (1984), 8–19.

[3] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.

Listing 1 Compression algorithm (pseudocode)

```
While not end of file
   Read next block of data into buffer and
      enter all pairs in hash table with counts of their occurrence
   While compression possible
      Find most frequent byte pair
      Replace pair with an unused byte
      If substitution deletes a pair from buffer,
         decrease its count in the hash table
      If substitution adds a new pair to the buffer,
         increase its count in the hash table
      Add pair to pair table
   End while
   Write pair table and packed data
End while
```

Listing 2 Expansion algorithm (pseudocode)

```
While not end of file
   Read pair table from input
   While more data in block
      If stack empty, read byte from input
      Else pop byte from stack
      If byte in table, push pair on stack
      Else write byte to output
   End while
End while
```

Listing 3 Compression program

```
/* compress.c */
/* Copyright 1994 by Philip Gage */

#include <stdio.h>

#define BLOCKSIZE 5000   /* Maximum block size */
#define HASHSIZE  4096   /* Size of hash table */
#define MAXCHARS   200   /* Char set per block */
#define THRESHOLD    3   /* Minimum pair count */

unsigned char buffer[BLOCKSIZE]; /* Data block */
unsigned char leftcode[256];     /* Pair table */
unsigned char rightcode[256];    /* Pair table */
unsigned char left[HASHSIZE];    /* Hash table */
unsigned char right[HASHSIZE];   /* Hash table */
unsigned char count[HASHSIZE];   /* Pair count */
int size;        /* Size of current data block */

/* Function prototypes */
int lookup (unsigned char, unsigned char);
int fileread (FILE *);
void filewrite (FILE *);
void compress (FILE *, FILE *);

/* Return index of character pair in hash table */
/* Deleted nodes have count of 1 for hashing */
int lookup (unsigned char a, unsigned char b)
{
  int index;

  /* Compute hash key from both characters */
  index= (a ^ (b << 5)) & (HASHSIZE-1);

  /* Search for pair or first empty slot */
  while ((left[index[ != a || right[index] != b) &&
        count[index] != 0)
    index = (index + 1) & (HASHSIZE-1);

  /* Store pair in table */
  left[index] = a;
  right[index]= b;
  return index;
}
```

```
/* Read next block from input file into buffer */
int fileread (FILE *input)
{
  int c, index, used=0;

  /* Reset hash table and pair table */
  for (c =  0; c < HASHSIZE; c++)
    count[c] = 0;
  for (c = 0; c < 256; c++) {
    leftcode[c] = c;
    rightcode[c] = 0;
  }
  size= 0;

  /* Read data until full or few unused chars */
  while (size < BLOCKSIZE && used < MAXCHARS &&
        (c = getc(input)) != EOF) {
    if (size > 0) {
      index = lookup(buffer[size-1],c);
      if (count[index] < 255) ++count[index];
    }
    buffer[size++] = c;

    /* Use rightcode to flag data chars found */
    if (!rightcode[c]) {
      rightcode[c] = 1;
      used++;
    }
  }
  return c == EOF;
}

/* Write each pair table and data block to output */
void filewrite (FILE *output)
{
  int i, len, c = 0;

  /* For each character 0..255 */
  while (c < 256) {

    /* If not a pair code, count run of literals */
    if (c == leftcode[c]) {
      len = 1; c++;
      while (len<127 && c<256 && c==leftcode[c]) {
        len++; c++;
      }
      putc(len + 127,output); len = 0;
      if (c == 256) break;
    }

    /* Else count run of pair codes */
    else {
      len = 0; c++;
      while (len<127 && c<256 && c!=leftcode[c] ||
          len<125 && c<254 && c+1!=leftcode[c+1]) {
        len++; c++;
      }
      putc(len,output);
      c -= len + 1;
    }

    /* Write range of pairs to output */
    for (i = 0; i <= len; i++) {
      putc(leftcode[c],output);
      if (c != leftcode[c])
        putc(rightcode[c],output);
      c++;
    }
  }

  /* Write size bytes and compressed data block */
  putc(size/256,output);
  putc(size%256,output);
  fwrite(buffer,size,1,output);
}

/* Compress from input file to output file */
void compress (FILE *infile, FILE *outfile)
{
  int leftch, rightch, code, oldsize;
  int index, r, w, best, done = 0;

  /* Compress each data block until end of file */
  while (!done) {

    done = fileread(infile);
    code = 256;

    /* Compress this block */
    for (;;) {

      /* Get next unused char for pair code */
```

```
for (code--; code >= 0; code--)
  if (code==leftcode[code] && !rightcode[code])
    break;

/* Must quit if no unused chars left */
if (code < 0) break;

/* Find most frequent pair of chars */
for (best=2, index=0; index<HASHSIZE; index++)
  if (count[index] > best) {
    best = count[index];
    leftch = left[index];
    rightch = right[index];
  }

/* Done if no more compression possible */
if (best < THRESHOLD) break;

/* Replace pairs in data, adjust pair counts */
oldsize = size - 1;
for (w = 0, r = 0; r < oldsize; r++) {
  if (buffer[r] == leftch &&
      buffer[r+1] == rightch) {

    if (r > 0) {
      index = lookup(buffer[w-1],leftch);
      if (count[index] > 1) --count[index];
      index = lookup(buffer[w-1],code);
      if (count[index] < 255) ++count[index];
    }
    if (r < oldsize - 1) {
      index = lookup(rightch,buffer[r+2]);
      if (count[index] > 1) --count[index];
      index = lookup(code,buffer[r+2]);
      if (count[index] < 255) ++count[index];
    }
    buffer[w++] = code;
    r++; size--;
  }
  else buffer[w++] = buffer[r];
}
buffer[w] = buffer[r];

/* Add to pair substitution table */
leftcode[code] = leftch;
rightcode[code] = rightch;
```

```
      /* Delete pair from hash table */
     index = lookup(leftch,rightch);
     count[index] = 1;
    }
    filewrite(outfile);
  }
}

void main (int argc, char *argv[])
{
  FILE *infile, *outfile;

  if (argc != 3)
    printf("Usage: compress infile outfile\n");
  else if ((infile=fopen(argv[1],"rb"))==NULL)
    printf("Error opening input %s\n",argv[1]);
  else if ((outfile=fopen(argv[2],"wb"))==NULL)
    printf("Error opening output %s\n",argv[2]);
  else {
    compress(infile,outfile);
    fclose(outfile);
    fclose(infile);
  }
}

/*End of File */
```

Listing 4 Expansion program

```
/* expand.c */
/* Copyright 1994 by Philip Gage */

#include <stdio.h>

/* Decompress data from input to output */
void expand (FILE *input, FILE *output)
{
  unsigned char left[256], right[256], stack[30];
  short int c, count, i, size;

  /* Unpack each block until end of file */
  while ((count = getc(input)) != EOF) {

    /* Set left to itself as literal flag */
  for (i = 0; i < 256; i++)
    left[i] = i;

  /* Read pair table */
  for (c = 0;;) {

  /* Skip range of literal bytes */
  if (count > 127) {
    c += count - 127;
    count = 0;
  }
  if (c == 256) break;

  /* Read pairs, skip right if literal */
  for (i = 0; i <= count; i++, c++) {
    left[c] = getc(input);
    if (c != left[c])
      right[c] = getc(input);
  }
  if (c == 256) break;
  count = getc(input);
}

/* Calculate packed data block size */
size = 256 * getc(input) + getc(input);

/* Unpack data block */
  for (i = 0;;) {

    /* Pop byte from stack or read byte */
    if (i)
      c = stack[--i];
    else {
      if (!size--) break;
      c = getc(input);
    }

    /* Output byte or push pair on stack */
    if (c == left[c])
      putc(c,output);
    else {
      stack[i++] = right[c];
      stack[i++] = left[c];
    }
  }
}
```

```
}

void main (int argc, char *argv[])
{
  FILE *infile, *outfile;
  if (argc != 3)
    printf("Usage: expand infile outfile\n");
  else if ((infile=fopen(argv[1],"rb"))==NULL)
    printf("Error opening input %s\n",argv[1]);
  else if ((outfile=fopen(argv[2],"wb"))==NULL)
    printf("Error opening output %s\n",argv[2]);
  else {
    expand(infile,outfile);
    fclose(outfile);
    fclose(infile);
  }
}
/* End of File */
```