High-Speed Data Compression

By Andy McFadden

KansasFest '93

Topics:

(1) LZW

(2) Defoliated Huffman

(3) LZSS

Source code credits:

"lzw.c" - part of "clzw.c", from lesson 8 of "Hacking Data Compression", a GEnie A2Pro Apple II University course.

"unpack.c" - part of "unpack.c", from the GNU gzip sources by Jean-loup Gailly.

"short-lzss.c" - drastically hacked piece of "lzss.c", originally by Haruhiko Okumura.

```
int
decode(insize)
long insize;
{
    unsigned char finalc;
    unsigned int entry;
    int incode, oldcode, ptr;

    at_bit = stack_idx = last_byte = oldcode = finalc = 0;
    entry = 0x101;
    table = real_table-256;

    while (1) {
        if (entry == 0x101) {
            if ((oldcode = incode = get_code(entry)) < 0) {
                fprintf(stderr, "ERROR: EOF hit early\n");
                return (-1);
            }
            if (incode == 0x100) return (0);    /* all done! */
            finalc = (oldcode & 0xff);
            putc(incode, outfp);
        }

        if ((incode = ptr = get_code(entry)) < 0) {
            fprintf(stderr, "ERROR: EOF Hit early\n");
            return (-1);
        }
        if (incode == 0x100) {            /* clear table code */
            entry = 0x101;                /* reset the table index */
            table = real_table-256;
            continue;                     /* back to top of while */
        }
        if (ptr > entry) {
            fprintf(stderr, "ERROR: invalid input\n");
            return (-1);
        }
        if (ptr == entry) {               /* handle KwKwK case */
            stack[stack_idx++] = finalc; /* last char is from previous phrase*/
            ptr = oldcode;                /* move down the tree */
        }

        /* move down the tree, stacking characters, until we reach a leaf */
        while (ptr > 0xff) {
            if (stack_idx >= 4096) {
                fprintf(stderr, "ERROR: stack overflow\n");
                return (-1);
            }
            stack[stack_idx++] = table[ptr].chr;    /* push it on */
            ptr = table[ptr].prefix;    /* move down the tree */
        }

        /* ptr is now < 0x100 */
        finalc = ptr;

        /* dump the stacked characters */
        putc(finalc, outfp);
        while (stack_idx)
            putc(stack[--stack_idx], outfp);
        table[entry].chr = finalc;
        table[entry].prefix = oldcode;
        entry++;
        oldcode = incode;
    }
}
```

```
/*
 * IN assertions: the buffer inbuf contains already the beginning of
 *   the compressed data, from offsets inptr to insize-1 included.
 *   The magic header has already been checked. The output buffer is cleared.
 */
int unpack(in, out)
    int in, out;            /* input and output file descriptors */
{
    int len;                /* Bit length of current code */
    unsigned eob;           /* End Of Block code */
    register unsigned peek; /* lookahead bits */
    unsigned peek_mask;     /* Mask for peek_bits bits */

    ifd = in;
    ofd = out;

    read_tree();      /* Read the Huffman tree */
    build_tree();     /* Build the prefix table */
    clear_bitbuf();   /* Initialize bit input */
    peek_mask = (1<<peek_bits)-1;

    /* The eob code is the largest code among all leaves of maximal length: */
    eob = leaves[max_len]-1;

    /* Decode the input data: */
    for (;;) {
        /* Since eob is the longest code and not shorter than max_len,
         * we can peek at max_len bits without having the risk of reading
         * beyond the end of file.
         */
        look_bits(peek, peek_bits, peek_mask);
        len = prefix_len[peek];
        if (len > 0) {
            peek >>= peek_bits - len; /* discard the extra bits */
        } else {
            /* Code of more than peek_bits bits, we must traverse the tree */
            ulg mask = peek_mask;
            len = peek_bits;
            do {
                len++, mask = (mask<<1)+1;
                look_bits(peek, len, mask);
            } while (peek < parents[len]);
            /* loop as long as peek is a parent node */
        }
        /* At this point, peek is the next complete code, of len bits */
        if (peek == eob && len == max_len) break; /* end of file? */
        put_ubyte(literal[peek+lit_base[len]]);
        skip_bits(len);
    } /* for (;;) */

    flush_window();
    if (orig_len != bytes_out) {
        error("invalid compressed data--length error");
    }
    return OK;
}
```

```
/*
 * lzss.c - extremely simple LZSS
 */
#include <stdio.h>

#define N           4096    /* size of ring buffer */
#define F             18    /* upper limit for match_length */
#define THRESHOLD      2    /* encode string into position and length
                               if match_length is greater than this */

unsigned char text_buf[N + F - 1];     /* ring buffer of size N,
              with extra F-1 bytes to facilitate string comparison */
int match_position, match_length;
FILE     *infile, *outfile;  /* input & output files */

/*
 * Find a match from the string at [r .. r+F-1] with the rest of the buffer.
 * Just a stupid linear search.
 */
void
FindMatch(r)
int r;
{
    unsigned char *key;
    int i, j;

    key = &text_buf[r];         /* string to match */
    match_length = 0;
    for (i = 0 ; i < N; i++) {
        if ((i >= r && i < r+F) ||
            (r >= N-F && i <= r && i < ((r+F) & (N-1)) ))
            continue;           /* don't start inside self */

        for (j = 0; j < F; j++)
            if (key[j] != text_buf[i+j]) break;

        if (j == F) {                   /* found full match */
            match_length = F;
            match_position = i;
            return;
        }
        if (j > match_length) {         /* found better match */
            match_length = j;
            match_position = i;
        }
    }
}

void
Encode()
{
    int i, len;
    int  c, r, s, last_match_length, code_buf_idx;
    unsigned char  code_buf[17], mask;

    code_buf[0] = 0;            /* flag bits for char vs posn/len pair */
    code_buf_idx = mask = 1;    /* code buffer index, bit position mask */

    r = N - F;                  /* place to put incoming data */
    s = 0;
    for (i = 0; i < r; i++)
        text_buf[i] = 0;        /* init dictionary to zeros */

    /* fill the read-ahead section, but be careful: file could be < 18 bytes */
    for (len = 0; len < F && (c = getc(infile)) != EOF; len++)
         text_buf[r + len] = c;

    if (!len) return;   /* nothing to compress! */

    do {
        FindMatch(r);   /* sets match_length and match_position */
        if (match_length > len) match_length = len;  /* match_length
            may be spuriously long near the end of text */

        /* output a single char or a len/posn pair */
        if (match_length <= THRESHOLD) {
            match_length = 1;  /* Not long enough match.  Send one byte. */
            code_buf[0] |= mask;  /* 'send one byte' flag */
            code_buf[code_buf_idx++] = text_buf[r];  /* Send uncoded */
        } else {
            code_buf[code_buf_idx++] = (unsigned char) match_position;
            code_buf[code_buf_idx++] = (unsigned char)
                    (((match_position >> 4) & 0xf0)
                | (match_length - (THRESHOLD + 1)));  /* Send position and
                    length pair. Note match_length > THRESHOLD. */
        }

        /* if we've got 8 items, output them */
        if ((mask <<= 1) == 0) {  /* shift mask left one bit, until zero */
            for (i = 0; i < code_buf_idx; i++)  /* Send at most 8 units of */
                putc(code_buf[i], outfile);     /* code together */
            code_buf[0] = 0;  code_buf_idx = mask = 1;
        }

        /* replace the parts of the string we matched with new stuff */
        for (i = 0; i < match_length && (c = getc(infile)) != EOF; i++) {
              text_buf[s] = c;          /* read new bytes */
              if (s < F - 1) text_buf[s + N] = c;  /* If the position is
                   near the end of buffer, extend the buffer to make
                   string comparison easier. */
              s = (s + 1) & (N - 1);  r = (r + 1) & (N - 1);
                   /* Since this is a ring buffer, increment the position
                      modulo N. */
        }
        /* correctly handle stuff when at end of input */
        while (i++ < match_length) {
              s = (s + 1) & (N - 1);  r = (r + 1) & (N - 1);
              len--;
        }
    } while (len > 0);  /* until length of string to be processed is zero */

    /* if there's some left over, send it now */
    if (code_buf_idx > 1)
        for (i = 0; i < code_buf_idx; i++) putc(code_buf[i], outfile);
}
```