Lempel-Ziv Compression:
Design

Lucais Sanderson

March 8, 2023

1 Introduction

The purpose of this program is to implement the LZ Compression algorithm. This algorithm saves space by taking sequences of characters repeated frequently and uses pointers to reference those repeated sequences.

Specifically, we will use a Trie ADT to organize the sequences of characters and a Word Table ADT for the decompression so we know what each code refers to each prefix.

2 Pseudocode

2.1 Compression

See section A.1 in the assignment document.

```
compress(infile, outfile):
    root = TrieCreate()
    curr_node = root
    prev_node = NULL
    curr_sym = 0
    prev_sym = 0
    # START_CODE is 1
    next_code = START_CODE
    while read_sym(infile, &curr_sym) is true:
        next_node = trie_step(curr_node, curr_sym)
        if next_node is not NULL:
            prev_node = curr_node
            curr_node = next_node
        else:
            write_pair(outfile, curr_node.code, curr_sym, bit-length(next_code))
            curr_node.children[curr_sym] = trie_node_create(next_node)
            curr_node = root
            next_code = next_code + 1
        if next_code is MAX_CODE:
            trie_reset(root)
            curr_node = root
            next_code = START_CODE
        prev_sym = curr_sym
    if curr_node is not root:
        write_pair(outfile, prev_code.code, prev_sym, bit-length(next_code))
        next_code = (next_code + 1) % MAX_CODE
    write_pair(outfile, STOP_CODE, 0, bit-length(next_code))
```

```
    flush_pairs(outfile)
```

2.2 Trie ADT

```
# Below is the structure for a Trie Node
TrieNode:
    # ALPHABET is 256, number of ASCII characters
    TrieNode *children[ALPHABET]
    code
```

```
TrieNode *trie_node_create(code):
    # might not need to allocate?
    node = allocate(sizeof(TrieNode))
    node.code = code
    for i in ALPHABET:
        node.children[i] = NULL
    return node
```

```
trie_node_delete(TrieNode n):
    free(n)
```

```
TrieNode *trie_create():
    root = allocate(sizeof(TrieNode))
    root = TrieNode
    root.code = EMPTY_CODE
    return root
```

```
trie_reset(TrieNode root):
    for i in ALPHABET:
        trie_delete(root.children[i])
        root.children[i] = NULL
```

```
trie_delete(TrieNode n):
    # base case
    if n == NULL:
        terminate
    # recursive case
    else:
        trie_delete(next child)
    # delete current node
    trie_node_delete(n)
    n = NULL
```

```
TrieNode *trie_step(TrieNode n, sym):
    for n's children:
        if child == sym:
            return n.children[sym]
    # NULL if invalid child
```

```
    return NULL
```

2.3 Decompression

See section A.2 in the assignment document.

```
decompression(infile, outfile):
    table = wt_create()
    curr_sym = 0
    curr_node = 0
    next_code = START_CODE
    while read_pair(infile, &curr_code, &curr_sym, bit-length(next_code)) is true:
        table[next_code] = word_append_sym(table[curr_code], curr_sym)
        write_word(outfile, table[next_code])
        next_code = next_code + 1
        if next_code is MAX_CODE
            wt_reset(table)
            next_code = START_CODE
    flush_words(outfile)
```

2.4 Work Table ADT

```
# Word structure
Word:
    # array of "symbols", a word
    *syms
    # length of the array syms
    len
```

```
# WordTable is an array of Words
# (sorry for C syntax, not sure how else to describe it)
typedef Word * WordTable
```

```
Word *word_create(*syms, len):
    Word w = allocate(sizeof(Word))
    # might need to copy memory instead of just setting a pointer to same location
    w.syms = syms
    w.len = len
    return w
```

```
Word *word_append_sym(Word *w, sym):
    alloc new block for new word, nw
    copy contents of w to nw
    nw.length = nw.length + 1
    add one more byte to nw.syms[]
    add sym to nw.syms[]
    return nw
```

```
word_delete(Word *w):
```

```
    if w isn't empty word:
        free w->syms
    free(w)
```

```
WordTable *wt_create():
    WordTable wt = allocate(sizeof(Word *) * MAX_CODE)
    # empty word. can be null, just accommodate in other functions.
    wt[EMPTY_CODE].syms = NULL
    return wt
```

```
wt_reset(WordTables *wt):
    # start at index 2 so as not to delete empty code
    for i in range(2(START_CODE), MAX_CODE):
        word_delete(wt[i])
        wt[i] = NULL
```

```
void wt_delete(WordTable *wt):
    word_delete(wt[EMPTY_CODE])
    free(wt[EMPTY_CODE])
```

2.5 I/O

```
FileHeader:
    # magic number is 0xBAADBAAC ... 32 bit unsigned
    magic
    # protection bits for infile to use when writing outfile
    protection
```

```
int read_bytes(int infile, uint8_t *buf, int to_read):
    # bytes to keep track of how many bytes are read
    # and which part or the array to continue reading
    bytes = 0
    # temp is for the current bytes read from read()
    temp
    while temp != 0 and bytes < to_read:
        temp = read (to_read - bytes) bytes into buf from infile
        bytes += temp
    return bytes
```

```
int write_bytes(int outfile, uint7_t *buf, int to_write):
    counter = 0
    # temp is for the current bytes written from write()
    temp
    while temp != 0 and counter < to_write:
        temp = write (to_write - counter) bytes into buf from infile
        counter += temp
    return counter
```

```
read_header(int infile, FileHeader *header):
    read in sizeof(FileHeader) into header
    swap endianness if byte order not little endian
    verify magic number
```

```
write_header(int outfile, FileHeader *header):
    write sizeof(FileHeader) to output file supplied from header
```

```
bool read_sym(int infile, uint8_t *sym):
    # terminal, char_buf, and char_index are static, global variables within i/o
    # if at beginning or end of buffer, fill it back up
    if char_index not 0 or BLOCK:
        terminal = read_bytes(infile, char_buf, BLOCK)
        char_index = 0
    if less than BLOCK bytes are read (terminal) and index is terminal:
        # (both could be 0 for example)
        return false
    # set character and increment
    *sym = char_buf(char_index)
    return true
```

```
write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen):
    # first write code with bitlen bits
    for i = 0, bitlen:
        # if current bit in code is a 1, set bit_buf as such
        if bitmask code with 1 << i:
            bit_buf[bit_index / 8] |= (1 << (bit_index / 8))
        bit_index = bit_index + 1
        # if buffer full, flush and reset
        if buffer full:
            flush_pairs(outfile)
            bit_index = 0
            reset_buffer(bit_buf)
    # now do the same but for sym, which will always be 8 bits
    for i = 0, 8:
        # if current bit in code is a 1, set bit_buf as such
        if bitmask code with 1 << i:
            bit_buf[bit_index / 8] |= (1 << (bit_index / 8))
        bit_index = bit_index + 1
        # if buffer full, flush and reset
        if buffer full:
            flush_pairs(outfile)
            bit_index = 0
            reset_buffer(bit_buf)
```

```
flush_pairs(int outfile):
    write_bytes(outfile, bit_buf, bit_index / 8)
```

```
# essentially inverse of write_pair()
bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen):
    code = 0
```

```
    sym = 0
    for i = 0, bitlen:
        if buffer full or empty (initial state):
            read_bytes(infile, bit_buf, BLOCK)
            bit_index = 0
        if bitmask current bytes in bit_buf with 1 << (bit_index/8):
            code |= 1 << i
        bit_index++
    for i = 0, 8:
        if buffer full or empty (initial state):
            read_bytes(infile, bit_buf, BLOCK)
            bit_index = 0
        if bitmask current bytes in bit_buf with 1 << (bit_index/8):
            code |= 1 << i
        bit_index++
    if code is STOP_CODE:
        return false
    return true
```

```
write_word(int outfile, Word *w):
    for i = 0, w->len:
        if at end of buffer:
            flush_words(outfile)
            char_index = 0
        char_buf[char_index] = w->syms[i]
        char_index++
```

```
flush_words(int outfile):
    write_bytes(outfile, char_buf, char_index)
```

3 Citations

- Darrel Long from the assignment document.
- Thanks to Audrey Ostrom for this template!