Design Document - Assignment 6 Lempel-Ziv Compression

Hsiang Yun Lu

March 12, 2023

1 Purpose of Programs

The two main programs will be the implementation of compression and decompression using the Lempel-Ziv Compression algorithm. The `encode.c` is a compression program that compresses any file, text, or binary. The `decode.c` program can decompress any file, text or binary, that was compressed with `encode`. The `trie.c` contains the implementation of the ADT and functions for tries. The `word.c` program contains the implementations of the ADT and functions for words and word tables. The `io.c` contains implementations of an I/O module to read or write through some buffer. The two main programs will use the libraries and modules to perform efficient compression and decompression.

2 Files To Be Included in Directory *asgn6*

- `trie.c` and `trie.h`
  - This is the implementation of the `TrieNode` ADT, including type construction and declarations in the header file and associated functions in the .c file.
- `word.c` and `word.h`
  - This is the implementation of the `Word` and WordTable ADT, including type construction and declarations in the header file and associated functions in the .c file.
- `io.c` and `io.h`
  - This is the implementation of all the I/O used in compression and decompression, including function declarations in the header file and associated functions in the .c file.
- `encode.c`

- This is the implementation of the compression process, including the main() function.

- `decode.c`
  - This is the implementation of the decompression process, including the main() function.
- `endian.h`
  - A header file includes the functions to check endianness and swap endianness.
- `code.h`
  - A header file that defines the code values used in the main programs.
- `Makefile`
  - This file compiles the programs and builds the `encode` and `decode` executable
- `README.md`
  - This file describes how to use all the programs and `Makefile`, including explanations on command-line options
- `DESIGN.pdf`
  - This file describes the design and design process for all the programs
  - Include pseudocode
- `WRITEUP.pdf`
  - Things learned in this assignment in detail

3 Design Process of `trie.c`

`trie.c` contains the functions to access or manipulate the `TrieNode` ADT. The functions support creating a new `TrieNode` or a root of a Trie, deleting one `TrieNode` or a whole branch of a Trie, resetting a Trie ,and stepping through a Trie to find the next child.

4 Design/Structure of `trie.c`

```
TrieNode *trie_node_create(uint16_t index)
    TrieNode *n
    n ← calloc(memory for a TrieNode)
    Check n is not NULL
    TrieNode *child[ALPHABET]
    for i = 0 to ALPHABET - 1
        child[i] ← NULL
    n->children ← child
    n->code ← index
    return n

void trie_node_delete(TrieNode *n)
    Free memory of n
    n ← NULL

TrieNode *trie_create(void)
    TrieNode *n ← create a TrieNode using trie_node_create()
    if not n
        return NULL
    return n
void trie_reset(TrieNode *root)
    for i = 0 to ALPHABET - 1
        if root->children[i] exist
            delete TrieNode root->children[i] using trie_node_delete()

void trie_delete(TrieNode *n)
    if n doesn't exist
        return void
    else if n exist
        for i = 0 to ALPHABET - 1
            if root->children[i] exist
                delete TrieNode root->children[i] using trie_delete()
            else
                delete TrieNode root->children[i] using trie_node _delete()

TrieNode *trie_step(TrieNode *n, uint8_t sym)
    if n->children[sym] doesn't exist
        return NULL
    return n->children[sym]
```

5 Design Process of `word.c`

`word.c` contains the functions to access or manipulate the `Word` and `WordTable` ADT. The functions support creating a new `Word` or a new `WordTable` (an array of `Word`), deleting one `Word` or a `WordTable`, resetting a `WordTable` and appending a symbol to a `Word` to form a new `Word`.

6 Design/Structure of `word.c`

```
Word *word_create(uint8_t *syms, uint32_t len)
    Word *w
    w ← calloc(memory for a Word)
    w->syms ← syms
    w->len ← len
    if not w
        return NULL
    return w

Word *word_append_sym(Word *w, uint8_t sym)
    if w is 0
        uint8_t *s[1]
        s[0] ← sym
        w->syms ← s
        w->len ← 1
    int8_t *s[w->len + 1]
    s[w->len] ← sym
    w->syms ← s
    w->len ← w->len + 1
    return w

void word_delete(Word *w)
    Free thr memory of w

WordTable *wt_create(void)
    WordTable *wt
    wt ← calloc(memory for a WordTable: MAX_CODE of Word)
    wt[EMPTY_CODE] ← word of 0 length
    return wt

void wt_reset(WordTable *wt)
    for i = 0 to MAX_CODE - 1
        wt[i] ← NULL
```

void wt_delete(WordTable *wt)

Free memory of `wt`

7 Design Process of `io.c`

`io.c` contains the functions to perform the I/O in the compression/decompression process. `read_bytes` and `write_bytes` are basic functions to ensure the read-in and written-out byte numbers are right. `read_header` and `write_header` are for read in and write out file header in `encode` and `decode`, respectively. They also check the endianness in the beginning. `read_sym`, `write_pair` and `flush_pairs` are used in `encode.c` to perform "read in one symbol at a time and check if all symbols are read," "write out a pair of code and symbol to output file through buffer."

8 Design/Structure of `io.c`

int read_bytes(int infile, uint8_t *buf, int to_read)

```
r ← 0
while r < to_read
        tmp ← read to_read bytes from infile to buf using read()
    if tmp <= 0
        break
    r ← r + tmp
return r
```

int write_bytes(int outfile, uint8_t *buf, int to_write)

```
w ← 0
while r < to_write
        tmp ← write to_write bytes from buf to outfile using write()
    if tmp <= 0
        break
    w ← w + tmp
return w
```

void read_header(int infile, FileHeader *header)

```
r ← read in FileHeader size of bytes from infile to header
if not little_endian
    swap header
```

```
void write_header(int outfile, FileHeader *header)
    if not little_endian
        swap header
    w ← write FileHeader size of bytes from header to outfile

bool read_sym(int infile, uint8_t *sym)
    if all bytes in the buffer has been read
        Read in a new block of bytes from infile
    *sym ← buffer[total_syms % block size]
    total_syms ← total_syms + 1
    if no more symbol to read from the buffer or infile
        return false
    return true

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
    code_rev ← code wrote from LSB in bitlen
    sym_rev ← sym wrote from LSB
    start_pos ← ((total_bits % (block size * 8)) % 8) + 1
    bit_idx ← bitlen
    buff_index ← ((total_bits % (BLOCK * 8)) / 8)
    cd_bi ← 0
    while bit_idx not 0
        mask ← bit mask for the bit at bit_idx position
        write_buff ← code_rev & mask
        cd_bi ← cd_bi | write_buff
        start_pos ← start_pos + 1
        bit_idx ← bit_idx - 1
        total_bits ← total_bits + 1
        if one byte is filled and should go to the next byte
            if whole block of buffer is filled
                buffer[buff_index] ← buffer[buff_index] | cd_bi
                flush_pairs(Write out everything in the buffer)
                buff_index ← 0
            else
                buffer[buff_index] ← buffer[buff_index] | cd_bi
                buff_index ← buff_index + 1
            cd_bi ← 0
        else
            if bit_idx is 0
                buffer[buff_index] ← buffer[buff_index] | cd_bi
                cd_bi ← 0
```

```
    sym_idx ← 8
    sm_bi ← 0
    while sym_idx not 0
        mask ← bit mask for the bit at sym_idx position
        write_buff ← sym_rev & mask
        sm_bi ← sm_bi | write_buff
        start_pos ← start_pos + 1
        sym_idx ← sym_idx - 1
        total_bits ← total_bits + 1
        if one byte is filled and should go to the next byte
            if whole block of buffer is filled
                buffer[buff_index] ← buffer[buff_index] | sm_bi
                flush_pairs(Write out everything in the buffer)
                buff_index ← 0
            else
                buffer[buff_index] ← buffer[buff_index] | sm_bi
                buff_index ← buff_index + 1
            sm_bi ← 0
        else
            if sym_idx is 0
                buffer[buff_index] ← buffer[buff_index] | sm_bi
                sm_bi ← 0
```

void flush_pairs(int outfile)

```
    write out all bytes in the buffer to outfile
    set all bytes of the buffer to 0
```

read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)

```
    if all bytes in the buffer has been read
        Read in a new block of bytes from infile
    start_pos ← ((total_bits % (BLOCK * 8)) % 8) + 1
    bit_idx ← bitlen
    buff_index ← ((total_bits % (BLOCK * 8)) / 8)
    cd_bi ← 0
    cd_by ← 0
    while bit_idx not 0
        cd_bi ← cd_bi | (read one bit)
        start_pos ← start_pos + 1
        bit_idx ← bit_idx - 1
        total_bits ← total_bits + 1
        if one byte is filled and should go to the next byte
            if whole block of buffer is filled
```

```
                read a new block of bytes from infile
                buff_index ← 0
            else
                buff_index ← buff_index + 1
            cd_by ← cd_by | (cd_bi «= bit_idx)
            cd_bi ← 0
    *code ← (cd_by | cd_bi) read from LSB
    sym_idx ← 8
    sm_bi ← 0
    sm_by ← 0
    while sym_idx not 0
        sm_bi ← sm_bi | (read one bit)
        start_pos ← start_pos + 1
        sym_idx ← sym_idx - 1
        total_bits ← total_bits + 1
        if one byte is filled and should go to the next byte
            if whole block of buffer is filled
                read a new block of bytes from infile
                buff_index ← 0
            else
                buff_index ← buff_index + 1
            sm_by ← sm_by | (sm_bi «= sym_idx)
            sm_bi ← 0
    *sym ← (sm_by | sm_bi) read from LSB
    if no more symbol to read from the buffer or infile
        return false
    return true

void write_word(int outfile, Word *w)
    sm_idx ← w->len
    while sm_idx not 0
        buffer[total_syms % BLOCK] ← w->syms[w->len - sm_idx]
        sm_idx ← sm_idx - 1
        total_syms ← total_syms + 1
    if the buffer is filled
        write out all the bytes in the buffer to outfile

void flush_words(int outfile)
    write out all bytes in the buffer to outfile
    set all bytes of the buffer to 0
```

9 Design Process of `encode.c`

`encode.c` contains the main() and is the main implementation of file compression. The program will first take in command-line options and open input and output files based on the given options. After writing the header to the output file, a Trie will be created. A few counters are used to keep track of the next available code, the previous trie node, and the previously read symbol. Next, construct the Trie and write out the compressed pairs based on the symbols read in. Finally, flush out any unwritten, buffered pairs to the output file.

10 Design/Structure of `encode.c`

```
COMPRESS(infile, outfile)
    root ← trie_create()
    curr_node ← root
    prev_node ← NULL
    curr_sym ← 0
    prev_sym ← 0
    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_code)
            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_node.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)
```

11 Design Process of `decode.c`

`decode.c` contains the main() and is the main implementation of file decompression. The program will first take in command-line options and open input and output files based on the given options. After reading the header and verifying the magic number, a WordTable will be created. A few counters are used to keep track of current code and the next code. Next, complete the WordTable and write out the decompressed words based on the pairs read in. Finally, flush out any unwritten, buffered words to the output file.

12 Design/Structure of `decode.c`

```
DECOMPRESS(infile, outfile)    table ← wt_create()
    curr_sym ← 0
    curr_code ← 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)
```

13 Credit

- Dev's section on Feb. 28th.
- Part of the pseudocode from the *Asgn6.pdf* specifics in resources
- LSB binary operation concepts from the discussion