Lempel Zip Compression

Victor Rivera
Early Design Document

1 Lempel Zip Compression Design Goal

The goal of this project was to create two programs which perform Jacob Ziv's LZ78 compression and decompression algorithms. The program algorithms are lossless algorithms that allows users to compress and decompress text, binary files, and source code.

2 Design Overview

2.1 Program Files

This project contains five files used for encoding and decoding, several test scripts, a Makefile, this Design PDF, and a README.md.

2.1.1 Encoding/Decoding

The compressing and decompressing program files were all written in C and include the following:

- trie.c/trie.h
- word.c/word.h
- io.c/io.h
- code.h
- endian.h
- encode.c
- decode.c

2.1.2 Supporting Files

- Makefile (File used to create and delete encode and decode binary executables)
- test files (Several test scripts written in Bash)

2.1.3 Design Files

- README.md (File gives a summary of the project, its files, and how to run it)
- design.pdf (File gives a more in depth description of the project, its files, and how to run it)

2.2 Program Options

Note: By default, both decode and encode programs take input from stdin and output to stdout.

The encode program supports the following options:

- -i <input file> : specify input to compress.
- -o <output file>: specify output file to decompress.
- -v: display compression stats.

The decode program supports the following options:

- -i <input file> : specify input to decompress.
- -o <output file>: specify output file of decompressed input.
- -v: display decompression stats.

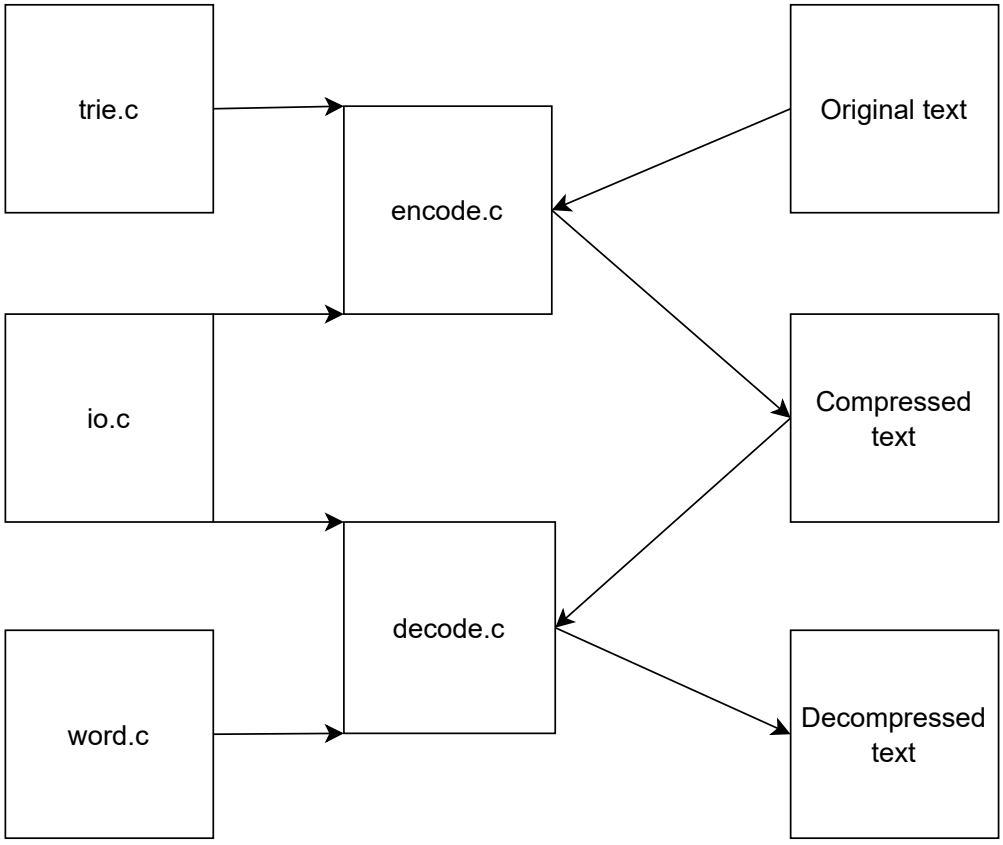

Figure 1: The block diagram above depicts the flow of how the program files interact with one another and with the file to compress. As shown, encode.c uses trie.c and io.c to compress the Original file and outputs the Compressed file. Similarly, decode.c uses word.c and io.c to decompress the Compressed file and outputs the Decompressed file. Made with draw.io [**?**].

3 Running The Program

To run the program, the encode and decode executables must first be created. To make the executables type "make" from the same directory as the Makefile. A data can then be compressed using the command "./encode". The -i ¡infile¿ and the -o ¡outfile¿ can be used to specify a file to compress and where to output the compressed data respectively. The -v option can be used to print out stats pertaining the encoding. Decoding a compressed file can be done using the command "./decode". The -i ¡infile¿ and the -o ¡outfile¿ can be used to specify the compressed file and the location to output the decompressed output respectively. Similarly to encode, the -v option can be used to output decompression stats. Encode and decode binary executables can then be removed by typing "make clean"

Summarized Instructions (options in "[]" are optional):

1. From the same directory as the Makefile type "make"
2. Encode: "./encode [-i ¡infile¿] [-o ¡outfile¿] [-v]"
3. Decode: "./decode [-i ¡infile¿] [-o ¡outfile¿] [-v]"
4. Clean: "make clean"

4 Compression/Decompression

4.1 TRIE - trie.c

The trie.c file contains functions used to create and manipulate a trie abstract data structure. A trie is a prefix tree, where each node contains n child nodes, 256 for this project. The trie (and its related functions) are used for the compression of files, since it is an optimal way of check for existing prefixes, or words.

4.1.1 trie.c Pseudocode

```
TrieNode data structure {
    Pointer to an array of 256 TrieNodes (TrieNode *children[256])
    code
}
```

```
trie_node_create(code){
    Allocate space the size of a trienode to a new trie node.
    for the children of the trie node (256 indixes) {
        trie->children[index] = NULL
    }
    trie_node->code = code
    return trie_node
}
```

```
trie_node_delete(trie_node){
    free memory allocated to trie_node
    return
}
```

```
trie_create(){
    Create a new trie node using trie_node_create with EMPTY_CODE as the input argument
    return a pointer to the created trie node
}
```

```
trie_reset(root){
    for the children of the root (256 indixes) {
        if the child node is not NULL{
            trie_reset(root->child_node[index]) // Recursive call using child node as input
            Free the node using trie_node_delete(root->child_node[index])
            Set the child node to NULL
        }
    }
    return
}
```

```
trie_delete(root) {
    for the children of the root (256 indixes) {
        if the child is not NULL{
            trie_delete(root->child_node[index]) // Recursive call using child node as input
        }
    }
    Free the node here to ensure all nodes get freed including the root
    return
}
```

```
trie_step(trie_node, sym){
    if the trie_node is not NULL{
        return trie_node's child at index sym (trie_node->child[sym])
    }
    Otherwise return NULL
}
```

4.2 WORD - word.c

The word.c file contains functions used to create and manipulate Word and WordTable data structures. A Word contains both a symbol string (words/prefixes) and a code corresponding to the string. A WordTable is an array of Words that starts with a Word containing an empty symbol string and START_CODE; it ultimately acts as a table of words and codes. Although a trie could have been used, word.c is used for decompression (decode.c) as it makes looking up code to word translations quicker, thus significantly reducing the time it takes to decompress a file.

4.2.1 word.c Pseudocode

```
Word data structure {
    character pointer syms
    length of syms
}
```

```
word_create(syms, len){
    Allocate space the size of a Word to a new_Word.
    if len is not 0{
        allocate len amount of uint8_t's to the new_Words syms attribute (new_Word->syms)
    }
    if len is 0 then allocate 1 uint8_t to the new_Words syms attribute
    copy the input syms to the new_Word syms attribute (copy syms to new_Word->syms)
    return new_Word
}
```

```
word_append_sym(word, sym){
    Create a copy_word of the input word using word_create and passing in words attributes as
    arguments (word_create(word->syms, word->len))

    if the syms attribute of word is not empty{
        reallocate memory to the syms attribute of copy_word by 1 uint8_t to make room for
        the sym to appended.
    }
    // No need to reallocate memory if syms attribute of word is empty since it was allocated
    // 1 uint8_t when it was created.
    increase the len attribute of copy_word by 1
    add sym to the last index of copy_word's attribute syms.
    return copy_word
}
```

```
word_delete(word){
    if words attribute syms is not pointing to NULL free syms
    free word
    return
}
```

```
wt_create(){
    Allocate memory for a word_table of (UINT16_MAX - 1) Words
    Create a word with empty sym ("") and len 0 and place it in word_table at index
    EMPTY_CODE

    return word_table
}
```

```
wt_reset(word_table){
    for all the words in the word table (UINT16_MAX - 1){
        if the Word is not NULL (if it exists) {
            Delete the word (word_delete())
            Set the word to NULL
        }
    }
    return
}
```

```
wt_delete(word_table){
    for all the words in the word table (UINT16_MAX - 1){
        if the word is not null delete the word (word_delete())
    }
    free memory allocated to the word table
    return
}
```

4.3 IO - io.c

The io.c file contains functions that are used for both compressing and decompressing files (encode.c and decode.c) to read and write from and to files. Reading and writing is done in blocks of 4KB to make the I/O more efficient. When reading, a new block of data is read in only when the previously read in data has been fully processed. Similarly, when writing, the buffered data is only written when the buffer is full.

4.3.1 io.c Pseudocode

```
struct FileHeader {
    uint32_t magic
    uint16_t protection
}
```

```
read_header(infile, header){
    read a block the size of a FileHeader into the input FileHeader "header"
    if the order of the bytes on the system is big endian{
        swap the protection attribute of the input header using (swap16(header->protection))
        swap the magic attibute of the input header using (swap32(header->magic))
    }
    return;
}
```

```
write_header(outfile, header){
    if the order of the bytes on the system is big endian{
        swap the protection attribute of the input header using (swap16(header->protection))
        swap the magic attibute of the input header using (swap32(header->magic))
    }
    write header to the outfile
    return;
}
```

```
read_sym(infile, sym){
    read a block of data from the input file to a buffer if the end of the buffer was reached
    or if we have never read into the buffer. Reset the index of the buffer.

    copy the symbol (byte) at the current index of the buffer into input sym
    increase the index of the buffer

    if sym is 0 then return false
    otherwise return true
}
```

```
buffer_pair(outfile, code, sym, bitlen){
    for (int i = 0; i < bitlen; i++ (length of code)){
        if the bit_index/8 is at the buffer limit{
            write the buffer out to the output file
            reset the buffer
            reset the bit_index
        }
        check if the bit of the input code at index i is set.
        if the bit is set{
            set the bit of the buffer at index (buffer[bit_index / 8] since bit_index is
            kept track of in bits not bytes)
        }
        increase the bit_index value by 1
    }

    for (int j = 0; j < CHAR_BIT; j++){
        if the bit_index/8 is at the buffer limit{
            write the buffer out to the output file
            reset the buffer
            reset the bit_index
        }
        check if the bit of the input sym at index j is set.
        if the bit is set{
            set the bit of the buffer at index (buffer[bit_index / 8] since bit_index is
            kept track of in bits not bytes)
        }
        increase the bit_index value by 1
    }
    return
}
```

```
flush_pairs(outfile){
    write the pair buffer to the outfile
    reset the buffer
    return
}
```

```
read_pair(infile, code, sym, bitlen){
    for (i = 0; i < bitlen; i++) {
        if the bit_index/8 is at the buffer limit or no blocks have been read{
            reset the buffer
            read in a new block from infile into the buffer
            reset the bit_index
        }
        check if the bit of the buffer[bit_index/8] at index i is set.
        if the bit is set{
            set the bit of input code at index i
        }
        increase the bit_index by 1
    }

    for (j = 0; j < CHAR_BITS; j++) {
        if the bit_index/8 is at the buffer limit or no blocks have been read{
            reset the buffer
            read in a new block from infile into the buffer
            reset the bit_index
        }
        check if the bit of the buffer[bit_index/8] at index j is set.
        if the bit is set{
            set the bit of input sym at index j
        }
        increase the bit_index by 1
    }

    if code is the STOP_CODE return false
    otherwise return true
}
```

```
buffer_word(outfile, word){
    for all the character of the word attribute sym {
        for (j = 0; j < CHAR_BITS; j++ (all bits in character)) {
            check if the bit of the character is set
            if it is set{
                set the bit of buffer[bit_index/8] at index j
            }
        }
        increase the bit_index by one
        if the bit_index/8 reaches the buffer limit{
            write the buffer to the outfile
            reset the buffer
            reset the bit_index
        }
    }
    return
}
```

```
flush_word(outfile) {
    write the word buffer to the outfile
    reset the word buffer
    return
}
```

```
read_buffer(buffer, byte_num, infile){
    reset the buffer
    read byte_num bytes into the buffer from infile
    return
}
```

```
write_buffer(buffer, byte_num, outfile){
    write byte_num bytes of buffer into the outfile
    reset the buffer
    return
}
```

4.4 Encode - encode.c

"encode.c" is the primary file responsible for compressing files. It contains the core compression algorithm, leveraging "io.c" and "trie.c". Additionally, "encode.c" handles program options as well as file management.

4.4.1 encode.c Pseudocode

```
bitlen(x){
    if x is not 0{
        while x is 1{
           add one to count
           shift x to the left by 1
        }
        return count
    }
    else{
        return 1
    }
}
```

```
encode_header(outfile, infile){
    Allocate memory of one FileHeader to a new FileHeader.
    Add the magic attribute to the new FileHeader (new_FH->MAGIC)
    Add the protection header to the new FileHeader (permissions of the infile)
    write the header to the outfile
    free the memory allocated to the new header
    return
}
```

```
compress_file(root, outfile, infile){
    set curr_node to the root
    set curr_sym to 0
    while there are symbols in the infile{
        set next node to the output of trie_step(curr_node, curr_sym)
        if next node is NULL{
            set prev_node to curr_node and curr_node to next_node
        }
        else{
            write the pair (code and sym) to the outfile using buffer_pair()
            Add a new child for curr_node at index curr_sym using next_code
            make the curr_node the root
            increase next_code by 1
        }
        if next code has reached its limit (MAX_CODE){
            reset the trie (trie_rest)
            set curr_node to the root
            set the next_code to the START_CODE
        }
        set prev_sym to curr_sym
    }

    if curr_node is not the root{
        output the last pair in the file
        increase next_code by 1
    }
    Output the STOP_CODE to the outfile
    flush the pair buffer
}
```

4.5 Decode - decode.c

"decode.c" is the primary file responsible for decompressing files. It contains the core decompression algorithm, leveraging "io.c" and "word.c". Additionally, "decode.c" handles program options as well as file management.

4.5.1 decode.c Pseudocode

```
bitlen(x){
    if x is not 0{
        while x is 1{
           add one to count
           shift x to the left by 1
        }
        return count
    }
    else{
        return 1
    }
}
```

```
decode_header(infile){
    Allocate memory of one FileHeader to a new FileHeader.
    read the header from the infile to the new FileHeader.

    if the header magic does not match the expected MAGIC{
        free the new header
        print an error message and exit the program
    }
    free the header
    return
}
```

```
decompress_file(table, outfile, infile){
    set next_code to START_CODE
    while there are pairs in the infile{
        Add the appended symbol/word to the table at index next_code
        increase next_code by 1
        if next code has reached the code limit (MAX_CODE){
            reset the table
            set the next_code to the START_CODE
        }
    }
    flush the word buffer to the outfile
    return
}
```

4.6 Endian - endian.h

Code provided by Professor Darrell Long. This file contains functions to check if the order of bytes of a system is big or little endian, as well as functions for swaping the endianness of uint16_t, uint32_t, and uint64_t.

4.6.1 endian.h Code

```
is_big(){
    union{
        uint8_t bytes[2];
        uint16_t word;
    }test;
    test.word = 0xFF00;
    return test.bytes[0];
}
```

```
is_little(){
    return ! is_big();
}
```

```
static inline uint16_t swap16(uint16_t x){
    uint16_t result = 0;
    result  |= (x & 0x000000FF) << 24;
    result  |= (x & 0x0000FF00) << 8;
    result  |= (x & 0x00FF0000) >> 8;
    result  |= (x & 0xFF000000) >> 24;
    return  result;
}
```

```
static inline uint32_t swap32 (uint32_t x){
    uint32_t result = 0;
    result |= (x & 0x000000FF) << 24;
    result |= (x & 0x0000FF00) << 8;
    result |= (x & 0x00FF0000) >> 8;
    result |= (x & 0xFF000000) >> 24;
    return result;
}
```

```
static  inline  uint64_t swap64(uint64_t x){
  uint64_t  result = 0;
  result  |= (x & 0x00000000000000FF) << 56;
  result  |= (x & 0x000000000000FF00) << 40;
  result  |= (x & 0x0000000000FF0000) << 24;
  result  |= (x & 0x00000000FF000000) << 8;
  result  |= (x & 0x000000FF00000000) >> 8;
  result  |= (x & 0x0000FF0000000000) >> 24;
  result  |= (x & 0x00FF000000000000) >> 40;
  result  |= (x & 0xFF00000000000000) >> 56;
  return  result;
}
```

4.7 Code - code.h

This file contains important codes that are used for the compression and decompression of files (word.c and trie.c).

Codes defined in code.h:

- STOP_CODE 0
- EMPTY_CODE 1
- START_CODE 2
- MAX_CODE UINT16_MAX

5 Testing

Note: The tests can be ran all at once using the test.sh script using the command "./test.sh" from the same directory as the test.sh file.

- test.sh (Main test file that can be used to run all other tests)

- small_text.sh (Test verifying the compression and decompression of a small text file)
- large_text.sh (Test verifying the compression and decompression of a the bible as a text file)
- small_binary.sh (Test verifying the compression and decompression of a small png file)
- large_binary.sh (Test verifying the compression and decompression of a large png file)
- valgrind_text.sh (Test checking the program for memory leaks while compressing and decompressing the bible as a text file)
- utils.sh (File that contains functions used in test.sh (to clean temporary files and check if files exist))

6 Other Files

6.1 Makefile

The Makefile is a file written in make file language and is used to create an encode and decode binary executable files. The executables are created using the command 'make' and removed using 'make clean', both must be run in the same directory as the Makefile.

6.2 README.md

The README.md file gives a brief overview of the program, its files, and how to run it.

6.3 design.pdf

The design.pdf is this file. It gives a more in depth explanation about the program, its files, and how to run it.