Assignment 6 Design: Lempel-Ziv Compression

Sep Nasiriany

Due Date: March 3, 2023

1 Description

This program will simulate file compression that is lossless by encoding and decoding files using a dictionary containing symbols and codes. Lossless compression is vital to computer science because it ensures that data that was compressed can be later recovered without loss. Lossy compression on the other hand means that data that gets compressed and loses information in the process of being encoded. Codes are 16 bit unsigned integers and codes get filled with a maximum value of $2^{16}-1$. Dictionaries correlate the symbol or the specific bytes with a This program uses Lempel-Ziv compression which involves making a symbol of the each sequence of unread bytes and making a code out of them. They will later get decoded with its dictionary of symbols and codes being simultaneously reproduced.

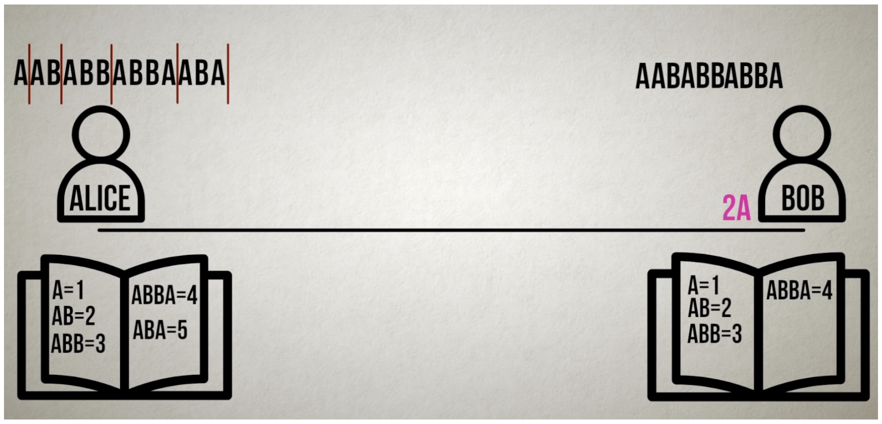

Figure 1
Diagram explaining a message from Alice to Bob being encoded and decoded. The books represent their dictionaries where the series of characters are the symbols and the number is equal to the code. Image from `https://www.youtube.com/watch?v=RV5aUr8sZD0`

There will be 5 C files that do each of the following:

1. `trie.c`: stores dictionary of symbols and words using nodes
2. `word.c`: does word translation during decompression

3. `io.c`: reads and writes data that gets buffered to encode and decode
4. `encode.c`: includes `main()` to encode
5. `decode.c`: includes `main()` to decode

2 `trie.c`

Tries are responsible for creating nodes for the message that is being given. A `TrieNode` struct will be made and it will have 2 fiels defined:

- `TrieNode *children [ALPHABET]`
- `uint16_t code`

The pointer to children will represent 256 pointers to trie nodes as children. The `code` field represents a 16 bit integer value that will store the combination for the bytes read.

2.1 `TrieNode *trie_node_create(uint16_t code)`

This is a Constructor for `TrieNode`. Its `code` is set to the parameter value `code`. All the children pointers are also set to `NULL`.

2.2 `void trie_node_delete(TrieNode *n)`

This is a Destructor for `TrieNode` and it will use the parameter `*n` and `free()` it.

2.3 `TrieNode *trie_create(void)`

This function creates a `trie` with the symbol `EMPTY_CODE` and returns the pointer if successful, otherwise returns `NULL`.

2.4 `void trie_reset(TrieNode *root)`

This function resets a `trie` by doing the following:

1. Run a `for` loop until we reach `MAX_CODES`
2. Delete the child nodes field by setting them to `NULL`

2.5 `void trie_delete(TrieNode *n)`

This function deletes a sub-trie from the node `n` recursively. Once that is done, set the pointer to the children array to `NULL` and call `trie_node_delete()`.

2.6 `TrieNode *trie_step(TrieNode *n, uint8_t sym)`

Returns a pointer to the child node with that represents the symbol `sym`. If there isn't a match, return `NULL`.

3 `word.c`

A `Word` is a struct that holds 2 fields:

- `uint8_t *syms`: Array of symbols
- `uint32_t len`: Length of the array `uint8_t *syms`

3.1 `Word *word_create(uint8_t *syms, uint32_t len)`

This is a Constructor for `Word` that sets the field `syms` to the parameter `syms` and the field `len` to the parameter `len`. If it successfully produces a word, it returns its pointer or `NULL` otherwise.

3.2 `Word *word_append_sym(Word *w, uint8_t sym)`

Constructs a new `Word` from the parameter `Word *w` and it adds a symbol `sym`.

1. Check if `Word *w` parameter is empty or `NULL`. If so, only append a symbol with an empty `Word`.
2. Return the new `Word` created

3.3 `void word_delete(Word *w)`

This is a Destructor for the `Word *w` using the `free()` to free memory allocated to it.

3.4 `WordTable *wt_create(void)`

This function creates a new `WordTable` which consists of an array of `Words`.

1. Create an array `WordTable` and initalize it.

2. The first `Word` in the array is the index `EMPTY_CODE` with a length of 0.

3.5 `void wt_reset(WordTable *wt)`

This function resets a `WordTable wt` to an empty `Word` and checks and sets other words are `NULL`.

3.6 `io.c`

The C file `io.c` will manage all I/O that is used for compression and decompression. It will use the system functions `read()` and `write()` in buffer arrays in order to greatly increase speed. It will read and write file headers that will verify and assign to ensure that compressed and decompressed files match each other.

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

This function is responsible for reading in 4KB blocks at a time and loading it into our buffer. This function will make sure to loop multiple times until it reads `to_read` bytes or `read()` returns 0, in which case we will break from the loop.

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

This function is responsible for writing out the buffer to outfile in 4KB blocks. Like the previous function, this function will have a do while loop that keeps going until it writes `to_read` bytes.

3.6.3 `void read_header(int infile, FileHeader *header)`

This function serves to read in each fileheader (the first 8 bytes in a file) and verify its magic number to see if it was encoded using the same program. It will do the following:

1. Call read_bytes() and pass in the `sizeof(FileHeader` for the number of bytes to read. This is 8 bytes.
2. If the system reads in big endian, make sure to swap the magic number and protection mask.
3. Verify the magic number by using an `assert()` statement.

3.6.4 `void write_header(int outfile, FileHeader *header)`

This function serves to write a fileheader for a new file. It will encode the protection mask and magic number. It does that by:

1. Check whether or not the system is in big endian. If so, swap the magic number and protection mask before writing to the file.

2. Call the `write_bytes()` function with the `sizeof(FileHeader` as its parameter for the number of bytes to read.

3.6.5 `bool read_sym(int infile, uint8_t *sym)`

This function serves read one symbol from the infile. It will do so with a global buffer of 4KB. It will do the following:

1. Check if sym index is 0, if so then fill up the buffer by calling `read_bytes()`.

2. If the buffer index has reached the end, then refill the buffer and set the index back to 0.

3. Return false if `read_bytes()` returns a value of 0.

4. Increment `total_syms` and `sym index`.

3.6.6 `void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)`

This function serves to write a pair (code, symbol) into outfile with a buffered array of 4KB. It will write it such that it is in LSB to MSB. It will do it by doing the following:

1. Create an index position variable that will indicate which bit index to start reading from. This will be calculated by taking `total_bits` mod 8.

2. Flip the first byte and use that for to calculate the value of code.

3. Read from index bit position of that byte, calculate its value. Flip it back and store it to the buffer.

4. If there are more bits for code to read, make sure to make a loop that will calculate the remaining bits.

5. Once the value of code has been calculated, make sure to do the same for symbol. Symbol is 8 bytes so it can either read one full byte or two bytes partially.

3.6.7 `void flush_pairs(int outfile)`

This function serves to write any remaining pairs that the buffer has not yet wrote to outfile. This function is to be called after `write_pair()` is done. It will set pair index to 0 to reset the buffer.

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

This function serves to read all pairs written to infile and decode it back into symbols. It will do so with a global buffer of 4KB. The code will be a variable number of bits (`bitlen`) that can go up to 16 bits. The symbol will always be 8 bits. Here's how it will work:

1. Fill up the buffer for the first time if `total_bits` is equal to 0.
2. Create a binary array for code that have size `bitlen`.
3. Calculate the first byte to calculate using an index position that is calculated like the previous function.
4. Only increment the pair index if the first byte has been used up completely.
5. Increment `total_bits` and update pair index accoridngly.
6. If there are more bits to be read for code, go into a `while` loop that will calculate it until finished.
7. Calculate its decimal value from the binary array using a `for` loop.
8. Create a binary array for sym as well, of size 8.
9. If there are 8 bits to be read from one byte, then read one byte. Otherwise read 2 bytes and put their bit values in the binary array.
10. Convert the binary array into a decimal sym value the same way it was done for code.

3.6.9 `void write_word(int outfile, Word *w)`

This function writes a word to outfile. It gets called in decompression after `read_pair()` gets called. It will use a `for` loop to flush or write all the word's symbols once the sym global buffer gets full. It will also increment `total_syms` and `symIndex`.

3.6.10 `void flush_words(int outfile)`

This function serves to write out any remaining words not written in `write_word()`. It will call `write_bytes()` and write symIndex number of bytes. It will reset the buffer and set symIndex to 0 once done.