Version 2.0

Section 0: Overview

- We will be creating a lossless compression system based on the Leppel-Ziv compression algorithm, which in this case utilizes a k-ary Trie node system.
- For a word such as “abcabcabcabc”, we can see a common pattern amongst its characters, and using a hashtable or dictionary would be very inefficient for parsing its characters compared to a trie search with O(1) time complexity.
- We are accessing indices of an alphabet array for a character we already know, so the time spent searching is far reduced.
- An example of how this algorithm works is as follows:
  - Word: “abcabcabcabc”
  - starting codes: EMPTY = 1 and START= 2
  - Check “a”, first appearance, and created by EMPTY + “a”, so add the word and code pair to the trie, a branch off root (which has code of EMPTY). The code for a is START
  - Check b, first appearance, add b. Code is 3
  - Check c, first appearance, add c. Code is 4
  - Check a, it has been added, so onto the next character. Is “ab” in our trie? Oh, it's null, so we can add a node for “b”, a child of “a” with a code 5 to our trie, a child of the node “a”.
  - And so on.
- This is how we generate our Trie, which when decoded looks at the code and generates a word, and can access the node very quickly from the Trie.
- For massive amounts of words this compression algorithm will save space by noticing patterns and accessing the words from the trie, much faster than adding a new word each time it shows up and searching for it in a hashtable.
- At a low amount of words/file size, obviously the time/space saved won’t be too noticeable, but that’s not too important since the main purpose of data compression is compressing large files from data centers or online hosting

servers, or anything really. It’s actually good to have for small files and does no harm, so it should be used frequently.

<u>Section 1: Pseudocode</u>

Trie.c

- Trie_create(code) :
  - Allocate memory for a pointer to a Trienode t, and set to contain code
- Trie_node_delete(n)
  - Free n and set n to null
- Trie_create
  - Creating a trie is simple, all we do is create a trie_node with code EMPTY_CODE. All its children are already null and we all set.
- trie_reset(root)
  - For each child of root, call trie_delete and set the child to NULL. We don’t want to delete root, just reset the trie to root.
- Trie_delete(n)
  - First confirm that the node is not null. If not, then for each child of the node recursively call trie_delete until reaching a null node, then we can delete the nodes and retrace.
- Trie_step(n,sym)
  - Returns the nodes child containing an index sym, where the corresponding code is stored. Remember code is the value of n->code and sym is index representing the ascii symbol.

Word.c

- Word_create(syms,len)
  - Allocate memory for a pointer to a word wrd, which contains len of len and an array of syms,characters which we copy in from syms. Allocate memory for the array, of len len and each value a size of a uint8 because a character is represented by 8 bits.

- word_append_sym(w,sym)
  - Creates a new word of w->len and w->syms.
  - We want the syms array of new_word to hold an extra character, so we reallocate the syms array to be len of w->len + 1 and store a uint8.
  - Remember to set the new words len to len +1.
- word_delete(w)
  - Frees the syms array and then w. Remember to set to NULL accordingly.
- wt_create()
  - Returns a word table, an array of word pointers.
  - Set wt[EMPTY_CODE] to an empty word, aka a len of 0 and its first charter ascii 0.
- wt_reset(wt)
  - For each word in the word table starting at START_CODE, if the word is not null, then delete the word and set the wt index to null.
- wt_delete(wt)
  - Same as reset, except we start at wt[0]. Free the wt and set to null.

io.c

- read_bytes(infile, *buf, to_read)
  - Create 2 variables to keep track of total bytes read and current bytes read.
  - Set bytes read to the # of bytes read in from the read() syscall.
  - Increment total,to_read and buf (the pointer to the index of the buffer) by # of bytes read. Do this while read() returns any # above 0. Return total.
- write_bytes(outfile,*buf,to_write)
  - Same as read_bytes except use the write() syscall
- write_header(infile, *header)
  - First check if the file system is big endian by calling the provided big_endian() function and then swapping header->magic and protection to the correct bit arrangement if big_endian is true
  - Write the header bytes to the outfile so we can check to see if we were the ones that compressed the infile

when decompressing later. NOTE: must cast the header as a uint8_t pointer when writing/reading bytes because our write_bytes function has a uint8_t *buf as the buf parameter.

- read_header(infile,*header)
  - Read in the header bytes from the infile.
  - For the endianness same as write_header.
- Read_sym(infile, *sym)
  - Check if the global var index, which is for the symbol buffer, is == to 0, if so read in bytes from file and set var “end” to the # of bytes read in. Modify sym directly by dereferencing and setting *sym to to symbols[index].
  - Increment index by 1 and check if its equal to 4kb, if so then set index to 0.
  - Also, at the beginning of the function set an if statement to check if index == end and index != 0, which means less than 4k were read in.
- write_pair(outfile,code,sym,bitlen)
  - Check if system is big_endian, if so then call swap16 on code.
  - For each bit in code up to bitlen, which is the code’s threshold, set the bit in the corresponding location in the bit array if the code’s bit is 1, else set to 0.
  - If the bit index is at the end of the bit buffer, which we can find by dividing the bit_index by 8, then write out all the bytes in the bi buffer and set the bit index to 0.
  - Same procedure for the sym, except loop 8 times because a sym must be represented with 8 bits.
- flush_pairs(outfile)
  - Since we might not always write out all the bytes from the bit array as bit_index doesn’t reach the end, we need to flush the remaining byts to the outfile, we write_bytes up to bit_index / 8, or the number of occupied bytes in the bit index.
- read_pair(infile,*code,*sym,bitlen)

- Create a temp variable for code which we will use to modify accordingly after accessing the correct bits from the bit_index.
- For bitlen # of times, check the bit_buffer at bit_index for the bit and set the temp_code at i to the bit.
- If bit_index is at the end of the buffer, set to 0 and read more bytes into the buffer.
- Set the code to temp code, we can do this directly by *code = temp_code;
- Return false if no more bytes to read in, true if everything worked.
- Same concept for accessing syms.

- write_word(outfile)
  - For each character in a word, set the symbols buffer at index to the word's symbol at i
  - Increment index by 1 and check if its at the end of the buffer, in which we can write out the bytes from the sym buf if true.
- flush_words(outfile)
  - Write out the remaining bytes from the sym buf by writing bytes up to index from the sym buf.

encode.c

- Program arguments i,o,v are set accordingly if specified. Default input/outputs are stdin/stdout.
- Personally, I allocated the file header onto the heap. I then created a variable "statbuf", a stat struct which I can use to store the infiles permissions and modify the outfile with.
- Set header->protection to statbuf.st_mode and use fchmod() to modify the outfile with header->protection. Set the header->magic to MAGIC and write out the header to the outfile.
- The algorithm is implemented as follows
  - Create node variables root,cur_node,prev_node
  - Create variables for the curr/prev sym and a variable for the next_code, which begins at START_CODE.
  - While there are symbols to read in from the infile,

        - Set next_node to child of curr_node at index curr_sym
        - If the node is not null, which means the pair exists, prev_node = curr_node curr_node = next_node
        - Else write the current code/sym pair to outfile and set the current node’s child at curr_sym to a new node with a code of next_code
        - Reset curr_node to root and increment next code by 1.
        - If the code has reached the max_code, then we need to reset the trie, curr_node and next_code.
        - Set the prev_sym to the curr_sym
    - After the loop, If the current node isnt the root then we know we can write the pair. Increment next_code by the remainder of (next_code + 1) / Max_code
    - Write the STOP_CODE and 0 to the outfile so we know where we can stop when reading in bytes.
    - Delete the trie and free the header.
    - Implement the space_saving formula and print remaining info if v is specified.
    - Close files
    -

decode.c

- Program arguments i,o,v are set accordingly if specified. Default input/outputs are stdin/stdout.
- Allocate space for the file_header and and read in the header bytes from the infile into the header. Modify the outfile with header’s protection stats and check to see if the magic numbers match
- For the Algorthim:
    - Create a word_table which we will use for storing our words.
    - 3 variables, curr_code, curr_sym, next_code = START_CODE.
    - While reading in pairs into curr_code and curr_sym from the compressed file:
        - Set the wordtable of next_code to wt[current_code] appended with curr_sym.

        - Write the bytes to outfile.
        - Increment next_code and check if next_code has reached max_code. If so, we can reset the word_table and set next_code to start code again.
    - Flush remaining words to outfile.
    - Delete the wt and free the header
    - Print required statistics.