Viveka Agrawal
Professor Veenstra
CSE13S, Winter 2023

CSE13S ASGN6 DESIGN DOC

Description of the program:

The main idea behind the Lempel-Ziv compression algorithm is to represent repeated patterns in data using pairs which are each comprised of a code and a symbol. The code is an unsigned 16-bit integer and a symbol is an 8-bit ASCII character. We initialize the dictionary with an empty word of string length 0 at index EMPTYCODE (1). Next, we look at the first word in the input file and check if this word has already been seen before. Since this word does not exist in the dictionary, we store it at index STARTCODE (2) and store the previously seen word to be the empty word and output the pair (EMPTYCODE, 'a') to the output file. We repeat this process for the rest of the words in the infile. If the word does not exist in the dictionary, we store it at the next index in the dictionary and store the pair (code, symbol) in the outfile. Once all the words in the infile have been read-in, we output the final pair (STOPCODE, 0) to indicate the end of compression.

The Lempel-Ziv compression algorithm is most effective when the entropy in the data to be compressed is low. Since the algorithm represents repeated patterns in data using pairs, a higher number of repeated patterns in data is preferred. With a higher number of repeated patterns, the size of the compression dictionary will be smaller. As a result, the number of (code, symbol) pairs written to the outfile will be smaller and the compressed output file size will be much smaller than the uncompressed input file size. If the entropy is high, this means that there are a less number of repeated patterns in the data, so the size of the compression dictionary will be larger. A larger number of (code, symbol) pairs need to be written to the output file. As a result, the compression ratio will be smaller.

In this assignment, these source and header files must be created and submitted on git. They include:

encode.c: contains the main() function for the encode program.

encode.c supports the following command line options:
• -v : Print compression statistics to stderr.
• -i : Specify input to compress (stdin by default)
• -o : Specify output of compressed input (stdout by default)

decode.c: contains the main() function for the decode program.

decode.c supports the following command line options:
• -v : Print decompression statistics to stderr.
• -i : Specify input to decompress (stdin by default)
• -o : Specify output of decompressed input (stdout by default)

trie.c: the source file for the Trie ADT.

trie.h: the header file for the Trie ADT.

word.c: the source file for the Word ADT.

word.h: the header file for the Word ADT.

io.c: the source file for the I/O module.

io.h: the header file for the I/O module.

endian.h: the header file for the endianness module.

code.h: the header file containing macros for reserved codes.

Here are 2 other source and header files that I will be submitting as part of my code:

buffers.c: the source file which initializes 2 static 4KB uint8_t arrays.

buffers.h: the header file for declaring variables and the 2 static 4KB uint8_t arrays.

A makefile, readme document, and writeup must also be completed for this assignment.

Pseudocode:

trie.c

Create a function trie_node_create which has a parameter named code

- This is the constructor for a TrieNode.
- The node's code is set to code.
- Make sure each of the children node pointers are NULL.

Create a function trie_node_delete which has a parameter named n

- This is the destructor for a TrieNode.

Create a function trie_create which does not have a parameter

- Initializes a trie: a root TrieNode with the code EMPTY_CODE.
- Returns the root, a TrieNode, if successful, NULL otherwise.

Create a function trie_reset which has a parameter named root

- Resets a trie to just the root TrieNode.
- Reset the trie by deleting its children so that we can continue compressing/decompressing the file.
- Make sure that each of the root's children nodes are NULL.

Create a function trie_delete which has a parameter named n

- Deletes a sub-trie starting from the trie rooted at node n.
- This will require recursive calls on each of n's children.

- Make sure to set the pointer to the children nodes to NULL after you free them with trie_node_delete().

Create a function trie_step which has 2 parameters named n and sym

- Returns a pointer to the child node reprsenting the symbol sym.
- If the symbol doesn't exist, NULL is returned.

word.c

Create a function word_create which has 2 parameters named syms and len

- Constructor for a word where sysms is the array of symbols a Word represents.
- The length of the array of symbols is given by len.
- This function returns a Word if successful or NULL otherwise.

Create a function word_append_sym which has 2 parameters named w and len

- Constructs a new Word from the specified Word, w, appended with a symbol, sym.
- The Word specified to append to may be empty.
- If the above is the case, the new Word should contain only the symbol.
- Returns the new Word which represents the result of appending.

Create a function word_delete which has a parameter named w

- Destructor for a Word, w.

Create a function wt_create which has no parameters

- Creates a new WordTable, which is an array of Words.
- A WordTable is initialized with a single Word at index EMPTY_CODE.
- This Word represents the empty word, a string of length of zero.

Create a function wt_reset which has a parameter named wt

- Resets a WordTable, wt, to contain just the empty Word.
- Make sure all the other words in the table are NULL.

io.c

Create a function read_bytes which has parameters named infile, buf, and to_read

- This will be a useful helper function to perform reads.
- Write a wrapper function to loop calls to read() until we have either read all the bytes that were specified (to_read), or there are no more bytes to read.
- The number of bytes that were read are returned.
- This function is used whenever a read needs to be performed.

Create a function write_bytes which has parameters named outfile, buf, and to_read

- This function is very much the same as read_bytes(), except that it is for looping calls to write().
- We loop until we have either written out all the bytes specified, or no bytes were written.
- The number of bytes written out is returned.
- This function is used whenever a write needs to be performed.

Create a function read_header which has parameters named infile and header

- This reads in the size of FileHeader bytes from the input file.
- These bytes are read into the supplied header.
- Endianness is swapped if byte order isn't little endian.
- Along with reading the header, it must verify the magic number.

Create a function write_header which has parameters named outfile and header

- Writes size of FileHeader bytes to the output file.
- These bytes are from the supplied header.
- Endianness is swapped if byte order isn't little endian.

Create a function read_sym which has parameters named infile and sym

- An index keeps track of the currently read symbol in the buffer.
- Once all symbols are processed, another block is read.
- If less than a block is read, the end of the buffer is updated.
- Returns true if there are symbols to be read, false otherwise.

Create a function write_pair which has parameters named outfile, code, sym, and bitlen

- “Writes” a pair to outfile.
- In reality, the pair is buffered.
- A pair is comprised of a code and a symbol.
- The bits of the code are buffered first, starting from the LSB.
- The bits of the symbol are buffered next, also starting from the LSB.
- The code buffered has a bit-length of bitlen.
- The buffer is written out whenever it is filled.

Create a function flush_pairs which has a parameter named outfile

- Writes out any remaining pairs of symbols and codes to the output file.

Create a function read_pair which has parameters named infile, code, sym, and bitlen

- “Reads” a pair (code and symbol) from the input file.
- The “read” code is placed in the pointer to code.
- The “read” symbol is placed in the pointer to sym.
- In reality, a block of pairs is read into a buffer.
- An index keeps track of the current bit in the buffer.
- Once all bits have been processed, another block is read.
- The first bitlen bits are the code, starting from the LSB.
- The last 8 bits of the pair are the symbol, starting from the LSB.
- Returns true if there are pairs left to read in the buffer, else false.
- There are pairs left to read if the read code is not STOP_CODE.

Create a function write_word which has parameters named outfile and w

- “Writes” a pair to the output file.
- Each symbol of the Word is placed into a buffer.
- The buffer is written out when it is filled.

Create a function flush_words which has a parameter named outfile

- Writes out any remaining symbols in the buffer to the outfile.

encode.c

COMPRESS(infile, outfile)

```
1 root = TRIE_CREATE()
2 curr_node = root
3 prev_node = NULL
4 curr_sym = 0
5 prev_sym = 0
6 next_code = START_CODE
7 while READ_SYM(infile, &curr_sym) is TRUE
8       next_node = TRIE_STEP(curr_node, curr_sym)
9       if next_node is not NULL
10              prev_node = curr_node
11              curr_node = next_node
12      else
13              WRITE_PAIR(outfile, curr_node.code, curr_sym,
                BIT-LENGTH(next_code))
14              curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
15              curr_node = root
16              next_code = next_code + 1
17      if next_code is MAX_CODE
18              TRIE_RESET(root)
19              curr_node = root
20              next_code = START_CODE
21      prev_sym = curr_sym
22 if curr_node is not root
23       WRITE_PAIR(outfile, prev_node.code, prev_sym,
         BIT-LENGTH(next_code))
24      next_code = (next_code +1) % MAX_CODE
25 WRITE_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26 FLUSH_PAIRS(outfile)
```

decode.c

```
DECOMPRESS(infile, outfile)

1 table = WT_CREATE()
2 curr_sym = 0
3 curr_code = 0
4 next_code = START_CODE
5 while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code))
  is TRUE
6       table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
7       WRITE_WORD(outfile, table[next_code])
8       next_code = next_code + 1
9       if next_code is MAX_CODE
10              WT_RESET(table)
11              next_code = START_CODE
12 FLUSH_WORDS(outfile)
```

buffers.c

- Initialize all elements of the binary pair array to 0
- Set the index of the binary pair array to 0
- Initialize all elements of the character array to 0
- Set the index of the character array to 0
- Initialize a temporary buffer and the number of bits counted to 0
- Set the total number of symbols and bits to 0

buffers.h

- Declare the index of the binary pair array
- Declare the index of the character array
- Declare a temporary buffer
- Declare the number of bits counted
- Declare the binary pair array with a size of BLOCK
- Declare the character array with a size of BLOCK
- Create the function declaration to initialize the binary pair array and the character array