Assignment 6 DESIGN.pdf

Ananya Batra

March 12, 2023

1 Description

Purpose: Write a program to perform LZ78 compression on any text or binary file and a program to decompress any file compressed with the previous algorithm, respectively. Both programs operate on big and little endian systems.

Structure: A user can compress a file with **encode.c**, which implements LZ78 compression. LZ78 compression is a lossless compression that compresses a file by storing a sequence of bytes (words) into a dictionary that gets built during the compression process. Words are denoted by codes in the dictionary. The compression algorithm works by replacing any future occurrence of a word that exists in a dictionary with its code. These words are stored in a Trie ADT, implemented by **trie.c**. Each TrieNode contains a symbol and n child nodes (256 in our case). In addition to trie.c, encode.c also calls upon functions in **io.c**. For this assignment, both encode and decode programs perform read and writes in blocks of 4KB. io.c contains functions in order to facilitate efficient reading and writing. A user can decompress a file (compressed with encode.c) with **decode.c**. Decompression works by translating codes to words (using a lookup table called a Word Table which is just an array of words). The Word ADT is implemented in **word.c**. Makefile compiles all the c program files and builds the encode and decode executables.

2 Design

trie.c
trie_node_create(): TrieNode constructor; creates a node with the code set by the parameter code

- dynamically allocate memory for a TrieNode node
- set node's code = code
- loop through all the node's children and set them to null
- return node

trie_node_delete(): TrieNode destructor; frees all the memory allocated for a TrieNode

- if the node is not null, free the memory allocated for the node and set it equal to NULL

trie_create(): Initializes a trie by creating a root TrieNode with code EMPTY_CODE

- create root with trie_node_create() with code EMPTY_CODE
- if the root pointer is NULL, return NULL
- otherwise, return root

trie_reset(): resets a trie to only contain the root node

- if this doesn't work, loop through the root's children and call trie_delete on them
- set each child = NULL

trie_delete(): deletes a subtree (starting at the node n passed in as a parameter)

- if the node isn't null

    - loop through the node's children
    - call trie_delete() on each of the children

- call trie_node_delete on n

trie_step(): returns a pointer to the child node of a node with the value sym

- return n's child at index n

word.c
word_create(): Constructor for Word

- dynamically initialize memory for a Word word
- sets word's len = len
- dynamically allocate memory for word's syms. Create dynamically initialized array with len number of elements with each element being the size of a uint8_t.
- return word

word_append_sym(): Constructs a new Word which is the same as the Word passed in appended with sym

- dynamically initialize memory for a Word new_word
- set new_word's len = w− >len + 1
- loop through w− >len
    - new_word's sym at every index = w − >syms[i]
- set new_word − > syms at index w− >len = sym
- return new_word

word_delete(): frees all the memory allocated to a word

- free the memory allocated to w's syms

- free w

wt_create(): Creates a new word table

- initialize table as an array of word pointers
- if table is null, return null
- set the word at index EMPTY_CODE to an empty string of length 0 with word_create()
- return table

wt_reset(): resets a word table to only contain the empty word

- loop through all indices until MAX_CODE with i
  - if i is not the EMPTY_CODE, delete the word at index i with word_delete() and set the word at index i to null

io.c
read_bytes(): Helper function to perform reads. This function repeatedbly calls read() until the number of bytes specified are read or there are no more bytes left to read.

- initialize a variable total_bytes_read to keep track of all the bytes read and a variable bytes_read_in() to keep track of the bytes read in at each call of read()
- while total_bytes_read is less than the bytes to be read in
  - use read() to read in the bytes specified by to_read - total_bytes_read already from the infile into the buffer.
  - add the number of bytes read in to bytes_read_in
  - if bytes_read_in is 0, return the bytes_read_in
  - add bytes_read_in to total_bytes_read
- return bytes_read_in

write_bytes(): Helper function to perform writes. This function repeatedbly calls write() until the number of bytes specified are written out or there are no more bytes left to write.

- initialize a variable total_bytes_written to keep track of all the bytes written out and a variable bytes_written() to keep track of the bytes written out at each call of write()
- while total_bytes_written is less than the bytes to be written (to_write)
  - use write() to write out the bytes specified by to_write - total_bytes_written to the outfile from the buffer
  - add the number of bytes written to bytes_written
  - add bytes_written to total_bytes_written
- return total_bytes_written

read_header(): Populates the specified header from infile

- use read_bytes() to populate header from infile
- if the bytes are in big_endian ordering
  - use swap32() to swap the bytes for header's magic to make it little endian
  - use swap16() to swap the bytes for header's protection to make it little endian
- if header's magic is not the magic number specified (AKA the file was not compressed by the compression algorithm described later in this document), exit the program

write_header(): Writes out the header to outfile

- if the file is in big_endian ordering
  - use swap32() to swap the bytes for header's magic to make it little endian

    - use swap16() to swap the bytes for header's protection to make it little endian

- write the bytes to outfile

read_sym(): Sets sym = to the current sym in the buffer and returns whether there are any more symbols left to read

- initialize a variable end_of_buf = 0
- if this is the first time reading in (aka max_sym_index which is a global static variable is negative one), then set max_sym_index = to bytes read in with read_bytes() and set curr_sym_index = 0
- set sym = to the value at sym_curr_index in sym_buffer
- to figure out what to return:
- if sym_curr_index ¡ max_sym_index -1, increment sym_curr_index and return true because there's still more indices in the buffer to read in
- otherwise, if max_sym_index = BLOCK, there are a couple cases to take care of
    - set the buffer to 0 with mem_set and set sym_curr_index = 0
    - read in BLOCK bytes into sym_buffer from infile and set the bytes read equal to max_sym_index.
    - if max_sym_index is 0, return false because this means we're at the end of the buffer.
    - otherwise, return true because there are still more bytes to be read in
- otherwise if the current index is at the last index in the buffer and end_of_buf = 0 (this is for the case where the buffer is not full AKA we're at our last read)
    - increment sym_curr_index
    - increment end_of_buf
    - return true

- otherwise return false

write_pair(): a pair consists of a code followed by a symbol. This function sets the current bytes in the bit buffer to equal the code and symbol so that they can be written out when the buffer gets full

- loop through the bits of code from i = 0 to i = bitlen - 1
  - if the current index is greater than BLOCK * 8 (block is in bytes - multiplying it with 8 gets the number of bits in block)
    - write the buffer to outfile with flush_pairs()
  - check if the current bit in code is 1. The way to do this is the right shift code by i bits (to get the ith bit in the lowest bit position) and ANDing it with 1. If the result is 1, then the ith bit in code is a 1.
    - OR the bit_buffer[bit_curr_index/8] (to get the byte) with (1 left shifted by (bit_curr_index % 8)). This sets the byte's current index with respect to the current byte to 1
  - increment bit_curr_index
- do the same process for adding sym to the bit_buffer. This time, loop from i = 0 to 7 because sym is only 1 byte

flush_pairs(): writes out the bit buffer and resets it

- first initialize a variable called curr_index
- if bit_curr_index is divisible by 8, set curr_index = bit_curr_index/8. otherwise, set curr_index = bit_curr_index/8 + 1. This is because C does floor division. So when figuring out which byte curr_index is at, we need to add 1 to the result of bit_curr_index/8.
- write out the bytes from bit_buffer to outfile
- set add the bytes written out * 8 to total_bits
- set bit_curr_index to 0

- reset the buffer with memset()

read_pair(): sets the code and symbol by reading in the values into the bit buffer

- initialize code and sym as 0 since we only plan to be setting the bits that are 1
- if max_bit_index (global static variable initialized to -1) is -1 (AKA this is the first time we're reading from the bit_buffer)
  - read BLOCK bytes from bit_buffer and set max_bit_index to 8*(bytes read in) to get the number of bits read in
  - add the number of bits to total_bits
- loop from i = 0 to bitlen - 1
  - if bit_curr_index is equal to max_bit_index (which means we've gone through all the bytes in the bit_buffer already)
    - reset the buffer and set the bit_curr_index = 0
    - read in more bytes and set the number of bytes = max_bit_index * 8
    - add the number of bits to total_bits
  - check if the current bit in bit_buffer's current byte is 1. The way to do this is the right shift bit_buffer[bit_curr_index / 8] by (bit_curr_index % 8) bits (to get the current bit in the lowest bit position) and ANDing it with 1. If the result is 1, then the current bit in the bit_buffer is a 1.
    - if it is, OR *code with 1 left shifted by i
  - increment bit_curr_index by 1
- do the same process to set *sym
- return true if code is not STOP_CODE and false otherwise

write_word(): sets the current byte in the symbol buffer to be the current word and writes out the buffer when it gets full

- loop through the length of word with i
- if the current index of the buffer goes beyond the end of the buffer size (4096), write out the bytes in the buffer.
- the idea behind this is we want to fill the buffer with the symbols in word. Once the buffer is full, write out all the bytes
- set the element at the current index of the buffer = w-¿syms[i]
- increment current index of the buffer

flush_words(): writes out the sym buffer to outfile

- write the bytes from sym_buffer to outfile and add it to total_syms
- set sym_curr_index = 0
- reset sym_buffer

encode.c - main: contains the encode program as described at the top of this document

- parse through the commandline options using switch statements
  - 'i' opens the file specified to read the message to compress from
  - 'o' opens the file specified to write the compressed message to
  - 'v' sets the statistics flag to true
  - 'h' sets the help flag to true and will later help print help message
- create a stat struct called buffer
- obtain information about infile and store it into buffer with fstat
- dynamically initialize memory for a FileHeader pointer
- set header's magic and protection (get protection using st_mode)
- set the outfile's permission to match header's protections

- write the header to outfile
- Now, to compress (based on the pseudocode provided in the assignment sheet):
- create a root node and create a copy of it called curr_node. create a prev_node and next_code and initialize both to NULL
- initialize integers curr_sym, prev_sym, and next_code (initialized to START_CODE)
- Loop through all the symbols in infile with read_sym()
  - if next_node is not null (meaning current prefix has been encountered before), set prev_node = curr_node and curr_node = next_node
  - otherwise
    * write the pair to outfile
    * let curr_node's child at index curr_sym be a new trie_node whose code is next_code
    * set curr_node = root
    * increment next_code by 1
  - if next_code is the MAX_CODE
    * reset the trie
    * set curr_node = root
    * set next_code = START_CODE
  - set prev_sym = curr_sym
- free the header and close infile and outfile

decode.c - main: contains the decode program as described at the top of this document

- parse through the commandline options using switch statements
  - 'i' opens the file specified to read the compressed message from
  - 'o' opens the file specified to write the decompressed message to

  - 'v' sets the statistics flag to true
  - 'h' sets the help flag to true and will later help print help message
- initialize a FileHeader header and read the header from infile into the header
- set the permissions for the outfile as the header's protections
- create a new word table
- initialize curr_sym, curr_code, and next code (initialized to START_CODE)
- loop through all the pairs with read_pair()
  - table[next_code] = append the read symbol to the word denoted by the read code
  - write the word at index next_code in the word table to the outfile
  - if next_code = MAX_CODE, reset the word table aand set next_code = START_CODE
- flush any buffered words to outfile
- if statistics is enabled
  - set uncompressed_size = total_syms and initialize compressed_size to 0
  - if total_bits is divisible by 8, set compressed_size = total_bits / 8 + sizeof(FileHeader). otherwise, set compressed_size = total_bits / 8 + sizeof(FileHeader) + 1. This is because C does floor division. So when figuring out which byte total_bits is at, we need to add 1 to the result of total_bits/8 if it's not a perfect multiple of 8 (to round up).
  - calculate the percent difference with the formula: $(1 - compressed\_size/uncompressed\_size)$*100
  - print all three values to stderr
- delete the word table
- close infile and outfile

3 Credit

- I went to Yiyuan's section on 2/27 to understand the compression algorithm
- I went to the last 30 minutes of Audrey's group tutoring session on 2/27 to understand io.c
- I went to Eric's tutoring session on 2/28 to clear up some misconceptions I had about trie.c and on 3/2 to get some insight on read_pair()
- I went to Jessie's tutoring session on 3/6. She went over io functions and the discussion helped me fix my read_sym()
- I went to Miles' tutoring session on 3/7. He helped me fix my write_pair() and write_word(). Essentially, I was using the wrong index to set the byte in the buffer.
- I went to John's session on 3/7. He went over read_pair() and write_pair(). He helped me debug my read_pair() too.
- I utilized the pseudocode provided in the assignment sheet to write encode.c and decode.c