Cole Marquard
cdmarqua@ucsc.edu
11/29/2020

CSE13s Fall 2020
Assignment 7: Lempel-Ziv Compression

Design Document

The word behind compression comes from the latin word “comprimere” to squeeze together. This lab aims to do just that and squeeze together data into a more compact form that can then be uncompressed at will. Compression was made to be computationally heavy in place of memory heavy as files don't need to be used all the time. When a file isn't being used or needs to be sent over the internet, compression is used to speed up transmission and take up less space until it needs to be decompressed and used. This lab will be using the LZ78 compression/decompression algorithm to decompress and compress files of various sizes in efficient blocks of 4KB.
The inputs to our program are:

1. “-i x” as a command line argument will set the input file to x, default stdin
2. “-o x” as a command line argument will set the output file to x, default stdout
3. “-v” as a command line argument will print statistics on the decompression/compression inlc: Compressed/Decompressed file size, and the compression ratio
4. User input could be used by typing “< inputfile” after ./encode or ./decode ex: ./encode -i input.txt -o output.txt

Top Level:

The top level design of the code is given by the following pseudocode

Encode

Main:

Read Command line arguments

Open Files for reading and writing
Write Header()
Compression Algorithm// The encoding algorithm is explained later
Close Files

Decode

Main:

Read Command line arguments

Open Files for reading and writing
Read and Check Header
Decode Algorithm// The decoding algorithm is explained later
Close Files

Compression Algorithm:

```
root = TRIE_CREATE()
curr_node = root
prev_node = NULL
curr_sym = 0
prev_sym = 0
next_code = START_CODE
while READ_SYM(infile, &curr_sym) is TRUE
    next_node = TRIE_STEP(curr_node, curr_sym)
    if next_node is not NULL
        prev_node = curr_node
        curr_node = next_node
    else
        BUFFER_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
        curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
        curr_node = root
        next_code = next_code + 1
    if next_code is MAX_CODE
        TRIE_RESET(root)
        curr_node = root
        next_code = START_CODE
    prev_sym = curr_sym
if curr_node is not root
    BUFFER_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
    next_code = (next_code + 1) % MAX_CODE
BUFFER_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
FLUSH_PAIRS(outfile)
```

Compression Algorithm Explained:

The Algorithm starts out by creating a root trie which is a graph like data struct which connects nodes to children nodes and the children have their own children etc. Initially the root has not children and has a code variable of EMPTY_CODE to denote that it contains no words. A curr_node object will be instantiated, starting out as root, and will be used in order to traverse the trie while compressing. We will then need a counter to start at START_CODE that will increment as we go through the symbols within the file. We will also have two variables to keep track of the previous node and symbol being prev_node and prev_sym.

The loop

We will loop while read_sym() returns symbols for us to compress and the symbol read in will be curr_sym.

Then Set next_node to be trie_step(curr_node,curr_sym) to check if the trie has a child with the same symbol as curr_sym

If next_node is not NULL then there was a child in the trie with the same sym so set prev_node to curr_node and set curr_node to next_node

Else if next_node was NULL then there wasn't a child with the same symbol so we buffer the pair(curr_node->code,curr_sym) to a array of bits and add the symbol to the trie so set curr_node->children[curr_sym] to be a new trie node whos code is next code. Then reset curr_node to be the root and increment the value of next_code.

Check if next_code is equal to MAX_CODE and if it is use trie_reset() to reset the trie to just have the root as we only have a finite number of codes with 16 bits.

Update prev_sym to be curr_sym

Buffer the pair (STOP_CODE,0) so the decompression algorithm knows when to stop decompressing.

Flush_pairs to flush any unwritten, buffered pairs

Decompression Algorithm:

```
table = WT_CREATE()
curr_sym = 0
curr_code = 0
next_code = START_CODE
while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TRUE
    table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
    buffer_word(outfile, table[next_code])
    next_code = next_code + 1
    if next_code is MAX_CODE
        WT_RESET(table)
        next_code = START_CODE
FLUSH_WORDS(outfile)
```

Decompression Algorithm Explained:
Create a new word table with wt_create() and set all the entries to NULL. Set the 1 index of the table be just have and empty word “” and a length of 0. We will call this table.

Well have two uint16_t variables to keep track of the current_code and the next_code. Next code will start as START_CODE aka 1 and functions the same as a monotonic counter we used during compression which we also called next_code.

Read pair() is used to read in pairs from the input file and is looped through to get all the pairs in the binary input file. This function sets curr_code curr_sym to be what is read in by the function. This loop is stopped when the code STOP_CODE aka 0 is read in by Read pair().

We will append the read symbol with the word denoted by the read code and add the result and add the result to the table at the index of next_code. Aka adding a letter/symbol to the word at table[curr_code]. This is achieved by setting table[curr_code] equal to the output of word_append_sym().

Buffer the word we just constructed with buffer_word which adds the letters within the word we just made to an array of letters that will eventually be written to the output.

Increment next code and check if it equals MAX_CODE and if it has, reset the word table using wt_reset() and set next_code to START_CODE. Same as resetting the trie during compression.

Flush asy buffered words using flush_words().

Trie:

//create a node and return its pointer
trie_node_create(code):

```
    T = Allocate Memory for node
    T.code = code
    Return T
```

//used to free a single node, used later in trie
trie_node_delete(TrieNode *n) :

```
    free(n)
```

//used to create the root of the trie which has the code of EMPTY_CODE aka 1
***trie_create(void):**

```
    T = trie_node_create(EMPTY_CODE(aka 1))
    Return T
```

//used to reset the trie when the amount of codes>MAX_CODES as described earlier

void trie_reset(TrieNode *root) :

```
    //256 is used bc ascii has 256 symbols but can be  changed to other values if
using for binary data or hex etc,
    For x in range(256):
        If root->children[x] !=NULL:
            trie_reset(root->children[x])
            trie_node_delete(root->children[x])
```

This trie_reset function recursively calls itself to ensure that all children of children get deleted as it moves to each child node and deletes all of its children recursively.

void trie_delete(TrieNode *root) :

```
	//256 is used bc ascii has 256 symbols but can be  changed to other values if
using for binary data or hex etc,
	For x in range(256):
		If root->children[x] !=NULL:
			trie_delete(root->children[x])
	trie_node_delete(root)
```

This trie_delete function recursively calls itself to ensure that all children of children get deleted but unlike reset, the root itself gets deleted as well which ensures that no memory leaks happen within the data. The input root doesn't necessarily need to be the actual root of the trie but can be any node which you want its children and the node itself to be deleted.

Check to see if the node has a child at location sym in n->children
trie_step(TrieNode *n, uint8_t sym):

```
	If n->children[sym]!=NULL :
		Return n->children[sym]
	Else:
		return NULL
```

Used to see if there is a child with the symbol but if there isn't then the returned value is null to tell the compression algorithm to add a node.

Word:

```
	//allocate memory for a word, set values, and then return the pointer
	word_create(uint8_t *syms, uint32_t len):
		W = allocate memory for word
		W.syms = syms
		W.len = len
		Return W
```

***word_append_sym(Word *w, uint8_t sym):**

```
		NewStr[w.len+1] //create array size of w.len +1
		//set the first values to be the letters of the last word
		For x in range(w->len):
			NewStr[x] = w->syms[x]
//set the last letter to be the sym your appending
		NewStr[w->len] = sym
		*New_Word = word_create(NewStr, (w->len + 1))
		Return New_word
		//create the new word object and return the pointer
```

word_delete(Word w):

    //if there are letters to free, free them
    If w->len>1:
        free(w->syms)
    free(w)
    //then free the word itself.

wt_create(void):

Wt = Allocate memory for word table size 65535-1 aka max size of 16 bits unsigned

Wt[1] = word_create(“”,0)// set the first index of the word table to be an empty word of length 0 and then return the pointer to the object

Return Wt;

wt_reset(WordTable *wt):

    For x in range(65535-1):
        //set the len and word to nothing
        Wt[x]->syms = NULL;
        Wt[x]->len = 0;

wt_delete(WordTable *wt)

    For x in range(65535-1):
        If wt[x]!=NULL:
            word_delete(wt[x])
    free(wt)

//delete all the words and then free the word table itself

IO:

//loops syscall read until we read the amount we want or there is nothing left to read then returns the amount of bytes read.

int read_bytes(int infile, uint8_t *buf, int to_read):

//while read returns a value and the bytes we have read are not the amount we were told to read, continue to read bytes.

    While((c=read(infile,(buf+bytes_read), 1)!=0) &&bytes_read!=to_read):
            read_bytes+=1
    Return read_bytes

//loops syscall write until we write the amount we want or there is nothing left to write then returns the amount of bytes written.

read_bytes(int infile, uint8_t *buf, int to_write):

//while read returns a value and the bytes we have read are not the amount we were told to read, continue to read bytes.

While((c=write(infile,(buf+bytes_written), 1)!=0) &&bytes_written!=to_write):

bytes_written+=1

Return read_bytes

read_header(int infile, FileHeader *header):

read(infile, header, sizeof(header)) // reads in the header object as a whole object //as described within the struct I/O section allows us to check for the magic //number

//writes the object header itself to the file which would then be read in by //read_header()

write_header(int outfile, FileHeader *header) :

write(outfile, header, sizeof(header))

bool read_sym(int infile, uint8_t *sym) {

//check to see if we need to reload the buffer

if (IndexOfBuffer == 0 or IndexOfBuffer == 4000):

IndexOfBuffer = 0

clear_word_buff() //clear the buffer array bc we are adding new data

C = read_bytes(infile,word_buff,4000)

EndOfBuff = c //sets end of the buffer to the amount of data read in

//once there is no more data as the end of the buffer is change in the line “EndOfBuff = c” to something not 4000 which means we got here when index != 4000 but index=endofbuffer+1 so there is nothing left to read in.

if(IndexOfBuffer = EndOfBuff +1):

Return 0;

*sym = word_buff[IndexOfBuffer] //set pointer of sym equal to the next sym

//at index IndexOfBuffer in word buff

IndexOfBuffer+=1 //Increment the buffer

Return 1 // There was a value so we return true

Binary_buff[4000] initalize an array of length 4000

Binary_index = 0 //starting point for writing binary data

```
buffer_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bit_len) {
	//buffering the symbol
	For x in range(8);
		if(getBit(sym,x)>0): //if the LSB is set in sym
			setbit(Binary_buff,Binary_index )//set the Binary_index bit of the
			                                  //binary buffer

			Binary_index +=1 //increment the index
			If Binary_index >32000: //if we filed up the array
				write_bytes(outfille,binary_buff,4000)
				Binary_index =0 //start at 0 again for the new array
				clearBuffer()
	//buffering the code
	For x in range(bit_len);
		if(getBit(sym,x)>0): //if the LSB is set in the code
			setbit(Binary_buff,Binary_index )//set the Binary_index bit of the
			                                  //binary buffer

			Binary_index +=1 //increment the index
			If Binary_index >32000: //if we filed up the array
				write_bytes(outfille,binary_buff,4000)
				Binary_index =0 //start at 0 again for the new array
				clearBuffer()
flush_pairs(int outfile):
	 if (binary_index % 8 != 0) : // if it doesn't fit into a rounded number of
	                                     // bytes
		bytes_to_flush = binary_index / 8 + 1;
	else :
		 bytes_to_flush = binary_index / 8;

	write_bytes(outfile , Binary_buff,bytes_to_flush)
	binary_index = 0;
	clearBuffer();
```

//Loops through the binary array and gets the values of sym and code
read_pair(int infile, uint16_t *code, uint8_t *sym, uint8_t bit_len)

```
    if (bufferIndex == 0) : // if it just has been used or needs be used again
        clearBuffer();
        read_bytes(infile, Binary_buff, 4000)
    Start_index = bufferIndex
    Value =0 //used to store the value of the sym
    Offset = 0//used incase we have to reset the array mid check
    For x in range(8):
        If GetBit(Binary_buff,x+start_index-offset) >0 :
            value+=2^x
        bufferIndex+=1
        if (bufferIndex == 32000) { // if it needs to load in more
            offset = x
            clearBuffer()
            Read_bytes(infile, Binary_buff, 4000)
            bufferIndex = 0
            start_index = 0
            }
    *sym = value //setting the pointer of sym to the value we got
    start_index += 8 // increment to get the code

    Value2 =0 //used to store the value of the code
    Offset = 0//used incase we have to reset the array mid check
    For x in range(bit_len):
        If GetBit(Binary_buff,x+start_index-offset) >0 :
            value2+=2^x
        bufferIndex+=1
        if (bufferIndex == 32000) { // if it needs to load in more
            offset = x
            clearBuffer()
            Read_bytes(infile, Binary_buff, 4000)
            bufferIndex = 0
            start_index = 0
            }
    *code = value2
    If value2 == STOP_CODE:
        Return 0
    Else:
```

Return 1

buffer_word(int outfile, Word *w):

For x in range(w->len):

*(word_buf + word_index) = w->syms[x] //set the next letter in the word //buffer to be the next letter in the word given and repeat for all letters

word_index+=1;

If (word_index ==4000): //if the array of letters is full we write it out

write_bytes(outfile, word_buf, 4000)

clear_word_buff() //clear the array of words we just wrote

word_index = 0 //start from an index of 0 again

flush_words(int outfile):

write_bytes(outfile, word_buf, word_index)

Word_index = 0

Design Process

Over the course of this lab i modified my design multiple times

At first when starting this lab the whole idea of writing and reading from files was so beyond my capabilities as I never really wanted to work with files. But while trying to do this lab I think I gained a better understanding of how files work and how to interact with them.

Some of the issues I was running into were problems with reading the files themselves and parsing file data and I really had to break down what was happening and really understand the different components that went into IO in order to solve my problems.

The Trie and Word table portion of the lab were by far the easiest to understand, make, and implement into code which was pretty nice as they work as intended although I was getting a weird error where it would only work when I used printf but after some testing I found a way around the error through trial and error..

Overall this project was fairly challenging, not only to conceptualize but also in actual implementation of the code itself. My least favorite part of the project was easily the I/O portion as working with files is kind of a pain and only loading in a portion of the data at a time is even worse to deal with. It makes finding errors and coding in general a lot more of a hassle then I would want even my worst enemy to deal with. I think a general improvement would be to remove the efficient block reads and just have it encode and decode in general. I think learning about efficient block reads and writes is fun and good to know but only as a concept as the final product isn't very good in terms of actual compressors and I would have rathered spent more time trying to figure out a harder, more complex compression algorithm possibly a lossey algorithm instead of dealing with the efficient I/O as working with these files is kind of a nightmare.