Brady Chan

brmchan@ucsc.edu

12/13/2020

CSE 13s Fall2020

Lempel-Ziv Compression

Design Document

Encode.c/Decode.c

8.1 Compression

```
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              BUFFER_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        BUFFER_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
24        next_code = (next_code + 1) % MAX_CODE
25  BUFFER_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26  FLUSH_PAIRS(outfile)
```

This is the given encode function to encode the file. It takes in all symbols until there are no more with read sym and passes the sym and code to next_node. Depending on next node it determines whether or not to set change the nodes or buffer the pair. If the code ever hits the maximum, it is reset to zero and the program continues running. After its finished taking in and buffering the file it flushes whatever is left in the buffer to the outfile.

8.2 Decompression

```
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        buffer_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)
```

This is the given decompression code and it makes a word table and reads in pair of data from the infile. At the next code for the table it adds the new word and adds it to the buffer. Similarly to encode if it hits the max, the code is reset and table as well. Lastly, all the words that still remain are flushed to the out file.

```
Int main(int argv, char **argc) :
        Int c;
        Int infile;
        Int outfile;
        Char *infile_name
        Char *outfile_name
        Bool infile_bol = false
        Bool outfile_bol = false
        Bool stats = false
        while( getopt):
                Switch:
                Case v':
                        Stats = true
                        Break
                Case'i':
```

```
                infile_name = optarg
                Infile_bol = true
        Case ‘o’:
                Outfile_name =  optarg
                Outfile_bol = true
    If infile_bol:
        Infile = open(infile_name, O_RDONLY);
    If outfile_bol:
        Outfile = open(outfile_name, O_WRONLY|O_CREAT|O_TRUNC);
```

Given Code:

```
Uint8_t bitlen(uint16_t next_code) :
    Return (uint8_t)(log2 (next_code)) + 1
```

This is the beginning getopt of both encode and decode. Since they are both the same I didn’t bother writing it twice. In this I have to use a bool to remember that I have a name set to char pointer for both infile and optarg. This is done because this circumvents the make infer error. After I set permissions and open the file. Bitlen is a function to calculate the lower bound, log 2 + 1 so I don’t have to keep calling it throughout encode and decode.

Io.c

```
Int read_bytes(int infile, uint8_t *buf, int to_read):
    Int read = 2
    Int cnt = to_read
    Int total = 0
    While total != to_read && read != 0:
        Read = read(infile, buf+total, cnt)
        Total += read
        Cnt -= read
Int write_bytes(int outfile, uint8_t *buf, int to_write):
```

```
        Int write = 2
        Int cnt = to_read
        Int total = 0
        While tota != to_read && read != 0:
                write = write(infile, buf+total, cnt)
                Total += read
                Cnt -= read
Void read_header(int infile, fileheader *header):
        read_bytes(infile, header, sizeof(header))

Void write_header(int outfile, fileheader *header):
        write_bytes(outfile, header, sizeof(header))

Bool read_sym(int infile, uint8_t *bytes:
        Stat int read = 0
        If s_index = 0:
                Read = read_bytes:
        *bytes = sym_buff[s_index]
        s_index++
        if(s_index == 4096:
                S_index = 0
        If s_index > read:
                Return false
        Return true


Void buffer_pair(int outfile, uint16_t code, uint8_t sym, uint8_t bitlen):
        For i < bitlen:
                If code & (1 << i):
                        pair_buff[p_index/8] |= (1 << (p_index % 8))
                Else:
```

```
                pair_buff[p_index/8] &= ~(1 << (p_index % 8))
        P_index++
        If p_index/8 == 4096:
                P_index = 0
    For i < 8:
        If sym & (1 << i):
                pair_buff[p_index/8] |= (1 << (p_index % 8))
        Else:
                pair_buff]p_index/8] &= ~ (1<< (p_index%8))
        P_index++
        If 0_index/8 = 4096:
                P_index = 0

Void flush_pairs(int outfile):
    Int write:
    If p_index % 8 = 0:
        write = p_index/8
    Else
        Write = p_index/8 + 1
    If p_index != 0:
        write_bytes(outfile, pair_buff, write)
        P_index = 0

Bool read_pair(int infile, uint16_t *code, uint8_t *sym, uint8_t bitlen):
    For i < bitlen:
        If pair_buff_index = 0:
                Read_bytes
        If( pair_buff[p_index/8] >> (p_index % 8)) & 1:
                *code |= 1 << i
        Else
                *code &= ~ 1<< i
```

```
            If pair_buff_index/8 = 4096
                    Pindex = 0
        For i < 8:
            if  pair_buff_index = 0:
                    Read_bytes
            If  pair_buff[p_index/8] >> (p_index % 8)) & 1:
                    *sym |= 1 << i
            Else
                    *sym &= ~ 1 << i
            If pair_buff_index/8 = 4096
                    P_index = 0

Void buffer_word(int outfile, Word *w):
        For i < w->len:
            Sym_buff[index] = w->syms[i]
            Index++
            If index = 4096:
                    Write_bytes
                    Index = 0

Void flush_words(int outfile):
        If sym_buff_index != 0:
            write_bytes(
```

Read and write bytes work very similarly to each other, they both run through their respective files and read or write into or from the buffer. Read and write header also work by calling these two functions and using them to fulfill their jobs. Read sym reads from the infile and puts the characters into the buffer. It checks if the index is zero and refills the buffer if so. Else if the buffer index ever is larger than the amount of bytes read, it returns false because the function has gone beyond and there are no more symbols to read. Buffer pair shifts the 1 over to test each index of the code and sym. After it finds it, it adds it to the buffer, incrementing the index each time and making sure it doesn't go beyond the max index. Read pair works very similarly

however, instead of storing it into the buffer, the pairs are taken from the buffer and passed back by the pointer. Flush pair calculates the amount of bytes needed to be written and then calls the write bytes method to write out the buffer. Flush word calls write bytes as long as the buffer isn't empty.

Word.c

```
Word * word_create(uint8_t *syms, uint 32 len):
        Word *word = malloc
        word->syms = calloc len
        memcpy(word->syms, syms, len)
        word->len = len
        Return word

Word *word_append_sym(word * w, uint8t sym):
        If w->len:
                Word *newword = malloc
                newword->syms = calloc w->len + 1
                For i < w->len:
                        newword->syms[i] = w->syms[i]
                newword->syms[w->len] = sym
                newword->len = w->len + 1
                Return newword
        else:
                word_create(sym, 1)
Void word_delete(Word *w):
        Free w->syms
        Free w
WordTable *wt_create(void):
        WordTable *wt = calloc max_code
        Wt[empty_code] = calloc 1
```

```
Void wt_reset(WordTable *wt):
    For i < max_code:
        If wt[i]:
            Word_delete wt[i]

Void wt_delete(WordTable *wt):
    Wt_reset wt
    Free wt[empty_code]
    Free wt
```

Word create makes the mallocs the memory needed to store the size and then callocs a size to store inside the syms of the new word. After, the symbols are copied over and the len is set to the given argument. Append works similarly to word create, and uses the same method, however instead of using memcpy I use a for loop to run through the original word and store all the symbols in the new symbol. Then the last symbol is added to the end of the word. Else if the word given is null, it does call word create and makes a new word. Word delete frees memory allocated to hold the syms and frees the word. Wt create, creates the word table and callocs it equal to max_code because that is the max size. Wt_reset goes through all of the possible index in max_code and deletes the words. Wt_delete frees all word table so there are no memory leaks before freeing the empty_code and itself. This ensures that the word table class can deallocate all used memory without any valgrind errors.

Trie.c

```
TrieNode *trie_node_create(uint16 code):
    TrieNode *trie malloc
    trie ->code = code
    For i < ascii code:
        trie->children[i] = NULL


Void trie_node_delete(TrieNode *n):
    free(n)
```

```
TrieNode *trie_create(void):
    TrieNode *root = trie_node_create(empty_word);

Void trie_reset(TrieNode *root):
    For i < ascii code:
        If root->children[i] exists:
            trie_delete(root->children[i])

Void trie_delete(TrieNode *n):
    For i < ascii code:
        If n->children[i] exists:
            trie_delete(n->children[i])
    trie_node_delete(n)

TrieNode *trie_step(TrieNode *n, uint8_t sym):
    n->children[sym]
```

Trie allocates memory for itself and sets the code equal to the argument. Afterwords it runs through each index of the ascii setting it to NULL. trie_node_delete just has free to delete itself. Trie_create, creates the root at empty word by calling trie node create and passing it the empty word code. Trie reset runs through each possible asci char code and if it exists calls itself to use recursion and run down the tree. When its done and reached the end, it deletes the current node and returns. Lastly, trie_step just returns the node's children.

Development Process:

- At first I had a super big struggle with understanding exactly what each part of the lab was supposed to do. The instructions felt extremely confusing and didn't make much sense.
- The concept and implementation of the buffer was super straight forward, but the piazza posts and pdf always seemed to confuse me more.

- It was also terribly hard to troubleshoot and find out what functions were working and what wasn't. Troubleshooting was an extremely difficult challenge because it was difficult to pinpoint what was broken

What I Learned:

In this lab, I gained a better understanding of the purpose of buffers and how to implement them as well as immensely improved my troubleshooting skills. It was a challenge to understand at first but eventually it became much more clear how the use of buffers in io worked with trie and word. The program's initial test also caused some terrible bugs and helped improve my deduction and troubleshooting skills. As certain bytes or symbols would not print, or if the buffer wasn't working properly it was a huge learning experience to figure out how to individually test these parts.

Sources:

- Lab with Oly to help with design process and pseudo
- Piazza post to fix make infer error with open(), understanding the buffer system, and other aspects of the program
- Bv.c and bv.h inspiraction for bit shift