Mason Reali

Professor Thorn

CSE 13

12 December 2020

Design and Pseudo-code

The main rundown of this assignment was to implement the LZ78 compression and decompression algorithm from scratch, using only the provided pseudo-code for the algorithms and the header files for the required helper implementations. The basic flow for both encode and decode is to process the inputs from the command line, and if given an output or input, to open the file and make sure that the header matches the protection code in the case of decode, and in the case of encode to write the protection bits to the header of the file. After this the logic for encode and decode differ greatly.

For encode, a new trie is created, and the program begins to read through the provided infile letter by letter, and appending these letters to the root trie if they do not exist and extending the child trees if they do. These code/sym pairs are then written into a buffer and finally to the outfile.

For decode, this is essentially done in reverse. The program begins to read the code/sym pairs, and uses this data to reconstruct the words in a WordTable. Each word struct can be linked to another, essentially undoing the original trie struct. At this point the new words are written into a buffer, and eventually the outfile.

More detailed descriptions of each page of code can be found below, with its accompanying pseudo-code. Pseudo-Code is provided for all files with the exception of the provided header files and the MakeFile.

Encode.c

The encode function is the main file for the encode functionality of this program. The main function consists of three parts. The first is the receiving and processing of the command line arguments. The second handles opening and setting permissions for the files needed to process. The third is the actual LZ78 compression algorithm provided in the given python pseudo-code for this lab.

```
int main(int argc, char **argv)
#=============== Get Arguments ============
    bool stats = false
    bool givenIn = false
    bool givenOut = false
    char *infiler ="none"
    char *outfiler = "none"
    int d = 0
    while (User Inputs)
    switch (d):
            case 'v':
                stats = true
                break
            case 'i':
                givenIn= true
                infiler = optarg
                break
            case 'o':
                givenOut = true
                outfiler = optarg
                Break
#====================== Open Files =============
    int infile
    int outfile
    If givenIn:
        infile = open(infiler, O_RDONLY)
        if (infile < 0)

    else:
        infile = STDIN_FILENO

    If givenOut:
        outfile = open(outfiler, O_WRONLY|O_CREAT|O_TRUNC)
    else:
        outfile = STDOUT_FILENO
#====================== Verify Header =============

    class FileHeader h
    class stat temp
    fstat(infile,&temp)
    h.magic = 0x8badbeef
    h.protection = temp.st_mode
    write_header(outfile, &h)
#====================== Encode Logic =============
    TrieNode *root = trie_create()
    TrieNode *curr_node = root
    TrieNode *prev_node = NULL
    TrieNode *next_node
    uint8_t curr_sym = 0
    uint8_t prev_sym = 0
    uint16_t next_code = START_CODE
    int count = 0
```

```
while (read_sym(infile,&curr_sym)):
    count +=1
    next_node = trie_step(curr_node,curr_sym)
    if(next_node != 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 == MAX_CODE): // If reached max
        trie_reset(root)
        curr_node = root
        next_code = START_CODE

    prev_sym = curr_sym

if(curr_node != 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)
trie_delete(root)

If stats:
    print("Compressed file size: %.0f bytes\n", writes)
    print("Uncompressed file size: %.0f bytes\n", reads)
    print("Compression ration: %.2f%%\n",100*(1-(writes/reads)))
```

Decode.c

The encode function is the main file for the encode functionality of this program. The main function consists of three parts. The first is the receiving and processing of the command line arguments. The second handles opening and setting permissions for the files needed to process. The third is the actual LZ78 decompression algorithm provided in the given python pseudo-code for this lab.

```
int main(int argc, char **argv)
:
#=============== Get Arguments ============
    bool stats = false
    bool givenIn = false
    bool givenOut = false
    char *infiler ="none"
    char *outfiler = "none"
    int d = 0
    while (User Inputs)
    switch (d):
            case 'v':
                stats = true
                break
            case 'i':
                givenIn= true
                infiler = optarg
                break
            case 'o':
                givenOut = true
                outfiler = optarg
                Break
#====================== Open Files =============
    int infile
    int outfile
    If givenIn:
        infile = open(infiler, O_RDONLY)
        if (infile < 0)

    else:
        infile = STDIN_FILENO

    If givenOut:
        outfile = open(outfiler, O_WRONLY|O_CREAT|O_TRUNC)
    else:
        outfile = STDOUT_FILENO
/*====================== Verify Header =============*/
    struct FileHeader h
    read_header(infile, &h)
    if (h.magic != 0x8badbeef):
        printf("Magic Number Didnt Match!\n")
        return 0

/*====================== Encode Logic =============*/
    WordTable *table = wt_create()
    uint8_t curr_sym = 0
    uint16_t curr_code = 0
    uint16_t next_code = START_CODE
    while(read_pair(infile, &curr_code, &curr_sym, bit_length(next_code))):
        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 == MAX_CODE):
        wt_reset(table)
        next_code = START_CODE
flush_words(outfile)
wt_delete(table)
if(stats):
    printf("Compressed file size: %.0f bytes\n", reads)
    printf("Uncompressed file size: %.0f bytes\n", writes)
    printf("Compression ratio: %.2f%%\n",100*(1-(reads/writes)))
```

Word.c

The word functions handle the creation of the word table and word structs required by the LZ78 decompression algorithm. All writing and reading from the provided infile and outfile is done in 4kb chunks.

```
# ========= Creates a new word ==========
Word *word_create ( uint8_t *syms , uint64_t len )
    Word *w = (struct Word*) malloc(sizeof(struct Word))
    if (w == NULL)
        printf("w null\n")
        exit(0)
    w->syms = (uint8_t*)calloc(100,sizeof(uint8_t))
    if (w->syms == NULL)
        printf("asdasd\n")
        exit(0)

    w->syms = memcpy(w->syms,syms,sizeof(&w->syms))
    w->len = len
    return w


# ========= Appends a sym to existing word ==========
Word *word_append_sym ( Word *w, uint8_t sym )
    if (sym == 0):
       Return 0
    Word *aw = (struct Word*) malloc(sizeof(struct Word))
    int x = w->len
    if(aw == NULL):
        print("Memory allocation failed")
        exit(0)

    aw->syms = (uint8_t*)calloc(100,sizeof(uint8_t))
    if(aw->syms == NULL)
        printf("NULL!\n")
        exit(0)

    if(w->syms == NULL)
        printf("NULL!!\n")
        exit(0)

    memcpy(aw->syms,w->syms,sizeof(&aw->syms))
    aw->syms[x] = sym
    aw->len = x + 1
    return aw

# ========= Deletes a word ==========
void word_delete ( Word *w):
    free(w->syms)
    free(w)

# ========= Creates a new word table  ==========
WordTable *wt_create ( void )
    uint8_t empty_word[1] = :0
    WordTable *wt = calloc(MAX_CODE, sizeof(struct Word)):
    if (wt == NULL):
        printf("wt null\n")
        exit(0)
```

```
    wt[EMPTY_CODE] = word_create(empty_word,0)
    return wt


// ========= Resets a word table ==========
void wt_reset ( WordTable *wt):
    for (int i = 0; i<MAX_CODE; i++):
        if (wt[i]!=0):
            word_delete(wt[i])
            wt[i] = 0

// ========= Deletes a word table ==========
void wt_delete ( WordTable *wt):
    wt_reset(wt)
    free(wt)
```

Trie.c

The trie.c file is the functions required to create the trie stucture required for the LZ78 compression algorithm. The idea is to create a new tree by initializing the root trie, and to add to it by appending its list of children with pointers to new tries. This is what the LZ78 algortihm does to compress data.

```
# ========= Creates a starting Trie Node ==========
TrieNode *trie_node_create(uint16_t code):
    TrieNode *t = (struct TrieNode*) malloc(sizeof(struct TrieNode))
    //TrieNode *children[ALPHABET]
    t->code = code
    if (t->code == 0)
        printf("t null\n")
        exit(0)

    for(int i = 0; i<ALPHABET; i++):
        t->children[i] = 0

    return t



// ========= Deletes a trie node ==========
void trie_node_delete(TrieNode *n):

    free(n)


// ========= Creates a new Trie ==========
TrieNode *trie_create(void):
    TrieNode *t = (struct TrieNode*) malloc(sizeof(struct TrieNode))
    t->code = EMPTY_CODE
    if (t->code == 0):
        printf("t null\n")
        exit(0)

    for(int i = 0; i<ALPHABET; i++):
        t->children[i] = 0

    return t


// ========= Resets a trie ==========
void trie_reset(TrieNode *root):
    for(int i = 0; i<ALPHABET; i++):
        if (root->children[i] !=0):
            trie_delete(root->children[i])
            root->children[i] =0




// ========= Deletes an entire trie ==========
void trie_delete(TrieNode *n):
    for(int i = 0; i<ALPHABET; i++):
        if (n->children[i] !=0):
            trie_delete(n->children[i])


    trie_node_delete(n)
```

```
// ========= Steps through a trie ==========
TrieNode *trie_step( TrieNode *n, uint8_t sym ):
    if (n->children[sym] != 0):
        return n->children[sym]
      else:
        return NULL
```

io.c

Io.c is what deals with all of the reading and writing of this program. The first few functions deal with reading the header of the given infile or writing to it depending on if we are decoding or encoding. The next important function is the read_bytes function. This is really the workhorse of this entire file as it is what is solely responsible for every read that happens in this program. The next few functions are all encode/ decode specific. The names are clear on what most of the function do.

```
uint8_t BUFFER[BLOCK]
float reads
float writes
// ========= Reads Header ==========
void read_header (int infile , FileHeader * header )  :
    read(infile, header, sizeof(FileHeader))
    reads += sizeof(FileHeader)


// ========= Write header ==========
void write_header (int outfile , FileHeader * header )   :
    fchmod(outfile, header->protection)
    write(outfile, header, sizeof(FileHeader))
    writes += sizeof(FileHeader)


// ========= Reads from the infile ==========
int read_bytes(int infile)  :
    int rbytes = 0
    int total = 0
    int Need4Read = BLOCK
    int prev = 0
    while((rbytes = read(infile,BUFFER,Need4Read))>0)  :
        prev = rbytes
        total+=rbytes
        if(total == BLOCK)  :
            reads += BLOCK
            return -1
        Need4Read = BLOCK-total
    reads += prev
    return prev
int point = BLOCK
bool fail = true
int x
// ========= Returns symbols from buffer when called ==========
bool read_sym (int infile , uint8_t *sym )  :
    if (fail)  :
        if (point == BLOCK)  :
            x = read_bytes(infile)
            if (x>=0)  :
                fail = false

            point = 0


    *sym = BUFFER[point]
    point+=1
    if (fail==false && point-1 >= x)  :
        return false
```

```
    return true


// ========= Finds length of Int ==========
int bit_length(uint16_t code)  :
    return (int)log2(code)+1




uint8_t BUFFEROUT[BLOCK]
int point2
int loc = 7
// ========= Buffers a sym pair and code for writing ==========
void buffer_pair (int outfile , uint16_t code , uint8_t sym , uint8_t bitlen )  :
    int bit
    for (int i = 0; i<bitlen; i++)  :
        if(point2 == 4096 )  :
            write(outfile, BUFFEROUT, BLOCK)
            writes += BLOCK
            //memset(BUFFEROUT, 0, sizeof(BUFFER))   // It would be proper to clear the buffer
after each use, but it wont match the tests if i do
            point2 = 0
            loc = 7

        bit = (code >> i) & 1u
        if(bit == 1)  :
            BUFFEROUT[point2] |= 1UL << (7-loc)
         else  :
            BUFFEROUT[point2] &= ~(1UL << (7-loc))

        loc -=1
        if (loc == -1)  :
            point2+=1
            loc = 7


    for (int i = 0; i<8; i++)  :
        if(point2 == 4096 )  :
            write(outfile, BUFFEROUT, BLOCK)
            writes += BLOCK
            point2 = 0
            loc = 7

        bit = (sym >> i) & 1u
        if(bit == 1)  :
            BUFFEROUT[point2] |= 1UL << (7-loc)
         else  :
            BUFFEROUT[point2] &= ~(1UL << (7-loc))

        loc -=1
        if (loc == -1)  :
            point2+=1
            loc = 7



// ========= flushes whats left of the buffer ==========
void flush_pairs (int outfile )  :
    if (loc<7)  :
        point2 +=1
```

```
    writes += point2
    write(outfile, BUFFEROUT, point2 )

int lo = 7
// ========= Buffers a sym pair and code for reading ==========
bool read_pair (int infile , uint16_t *code , uint8_t *sym , uint8_t bitlen )  :
    int bit
    *code = 0
        if (point==4096)  :
            lo = 7
            point = 0
            x = read_bytes(infile)
            if (x>=0)  :
                fail = false
        if (fail==false && point== x-1)  :
            return false

        bit = (BUFFER[point] >> (7-lo)) & 1u
        if(bit == 1)  :
            *code |= 1UL << (i)
         else  :
            *code &= ~(1UL << (i))

        lo -=1
        if (lo ==-1)  :
            point+=1
            lo = 7


    *sym = 0
    for(int i =0; i<8; i++)  : // Now reads the sym that came with the code
        if (point==4096)  :
            lo = 7
            point = 0
            x = read_bytes(infile)
            if (x>=0)  :
                fail = false

        if (fail==false && point== x-1)  :
            return false

        bit = (BUFFER[point] >> (7-lo)) & 1u
        if(bit == 1)  :
            *sym |= 1UL << (i)
         else  :
            *sym &= ~(1UL << (i))

        lo -=1
        if (lo ==-1)  :
            point+=1
            lo = 7


    if (fail==false && point== x)  :
        return false

    return true

int yeet = 0
// ========= Buffers a word writing ==========
void buffer_word (int outfile , Word *w)  :
```

```
    for(uint32_t i =0; i<w->len; i++)  :
        if (yeet == BLOCK)  :
            writes += BLOCK
            write(outfile, BUFFEROUT, BLOCK)
            memset(BUFFEROUT, 0, sizeof(BUFFER))
            yeet = 0

        BUFFEROUT[yeet] = w->syms[i]
        yeet+=1



// ========= flushes remaining word ==========
void flush_words (int outfile )  :
    writes += yeet
    write(outfile, BUFFEROUT, yeet )
```