Aleesha Nageer
anageer@ucsc.edu
12/14/2020

CSE 13S Fall 2020
Assignment 7: Lempel-Ziv Compression
Design Document

The Lempel-Ziv Compression algorithm allows us to compress any file without losing any data (lossless). In this program, we are to write an algorithm that uses trie and word table data types to encode and decode different files. A trie breaks down the items in the file to branches of nodes that have patterns. These patterns are converted into symbols, which are what we use for compression. The word table can go through these symbols and translate them back to their original character, which is what we use for decompression.

The inputs to the program are:

-v: Prints statistics for the number of bytes in the compressed and the decompressed files. It also prints the ratio of file size for both files

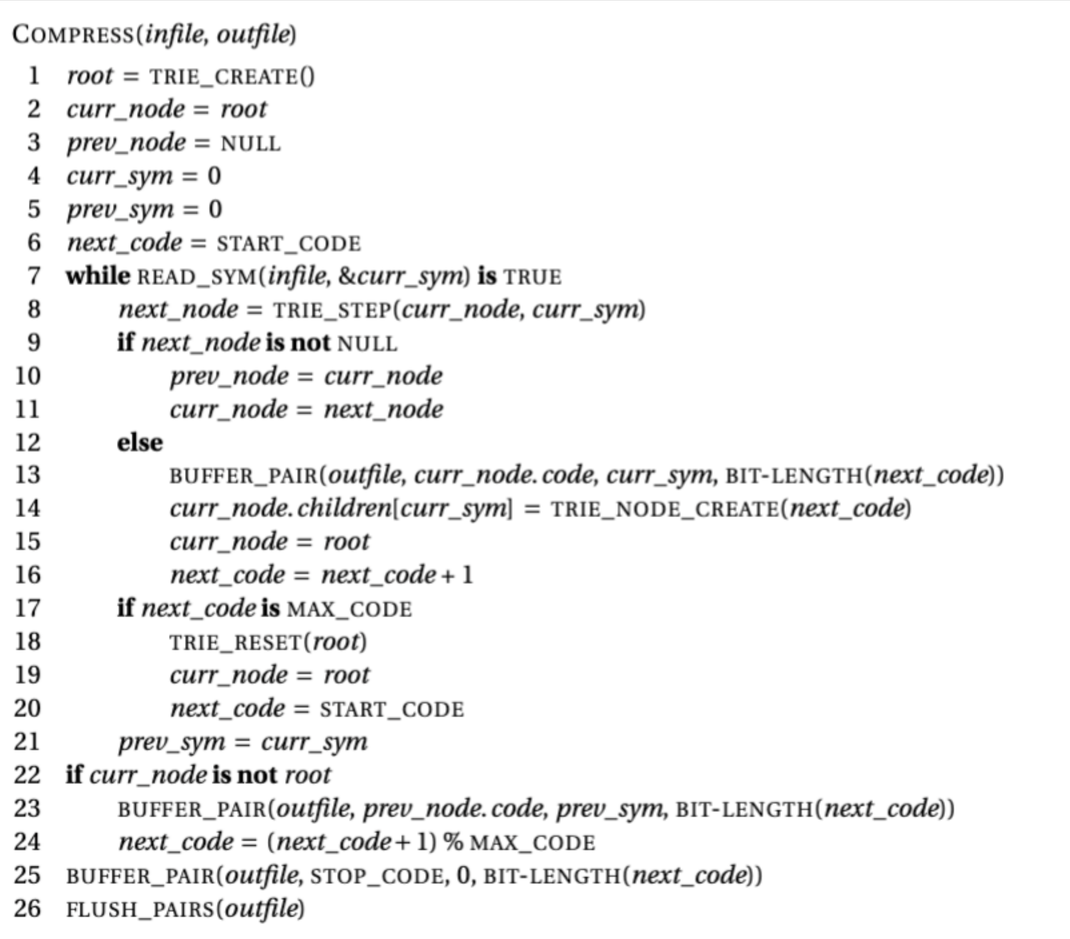
-i <file>: Sets the input file

-o <file>: Sets the output file

Top-Level

encode.c

```
Main:
    Read program arguments

    Create flags for switch statement

    Switch (method)
        Verbose
            Set statistics flag to 1
        Infile
            Set Infile
        Outfile
            Set Outfile

    If Statistics
        Print Compressed File Size
        Print Uncompressed File Size
        Print Compressed to Uncompressed Ratio

    Create Fileheader
    Set Magic Number
    Set Protection

    Write Header

    Use Provided Compression Algorithm
```

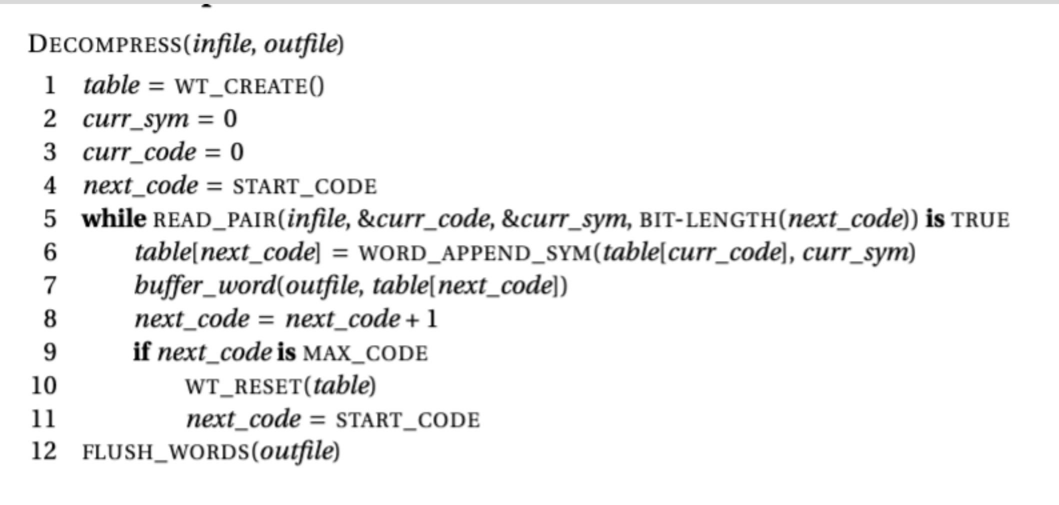


```
COMPRESS(infile, outfile)
root = TRIE_CREATE()
curr_node = root
prev_node = NULL
curr_sym = 0
prev_sym = 0
next_code = START_CODE
while READ_SYM(infile, &curr_sym) is TRUE
    next_node = TRIE_STEP(curr_node, curr_sym)
    if next_node is not NULL
        prev_node = curr_node
        curr_node = 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)
```

```
        Free Memory
```

decode.c

```
Main:
        Read program arguments

        Create flags for switch statement

        Switch (method)
                Verbose
                        Set statistics flag to 1
                Infile
                        Set Infile
                Outfile
                        Set Outfile

        If Statistics
                Print Compressed File Size
                Print Uncompressed File Size
                Print Compressed to Uncompressed Ratio

        Create Fileheader
        Read Header

        Check if Magic Numbers match
                If not return error

        Use Provided Decompression Algorithm
```



```
DECOMPRESS(infile, outfile)
    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)
```

trie.c

```
trie node create(code)
        create and allocate space for Trie Node
        set this node's code to code parameter

        return the node

trie node delete(node)
        free memory for node

trie create(void)
        create root trie node by calling trie node create with EMPTY_CODE parameter

trie reset(root)
        loop through trie node root's children
                free memory for children nodes

trie delete(node)
        loop through trie node root's children
        free memory for children nodes
    then free memory for root node

trie step(node, sym)
    returns symbol of word entered
```

word.c

```
word create(syms, len)
    creates word and allocates memory for it
    allocates memory for word’s symbols

    initialize word length
    fill word’s symbols with symbol parameter

    return w

word append sym(word, sym)
    if word exists
        allocate memory for new word
        allocate memory for appended symbol

        add symbols to new word
        append symbol

        increment length of new word
    else
        word create

word delete(word)
    free memory for symbols
    free memory for word

word table create(void)
    allocate memory for size of word table
    allocate memory and set word table indices to NULL

    return word table

word table reset(word table)
    iterate from start index to max index
    free memory of each word in the word table

word table delete(word table)
    iterate through entire word table
    free memory for everything in word table
```

io.c

```
static sym buffer, bit buffer
static sym index, bit index

read bytes(infile, buffer, bytes to read)
    initialize read bytes to 0
    initialize total read to 0
    do while read bytes is greater than 0 and total doesn’t equal bytes to read
        read bytes = read(infile, buffer + total, bytes to read – total)
        total += read bytes
    return total

write bytes(outfile, buffer, bytes to write)
    initialize written bytes to 0
    initialize total written to 0
    do while written bytes is greater than 0 and total doesn’t equal bytes to write
        written bytes = write(outfile, buffer + total, bytes to write – total)
        total += read bytes
    return total

read header(infile, file header)
    read bytes(infile, header, size of file header)

write header(outfile, file header)
    write bytes(outfile, header, size of file header)

read symbol (infile, byte)
    num read = 0
    boolean check

    if sym index is not 0
        num read = read bytes(infile, sym buffer, 4096)

    increment byte index

    if index reaches end
        send back to the beginning

    if 4096, there are bytes left to read
        check = true
    else
        if sym index = num read + 1
            check = false
        else
            check = true

    return check

buffer pair(outfile, code, sym, bit length)
    iterate from 0 to bit length
        if code anded with 1 = 1
            set bit
        else
            clear bit

        increment bit index
        shift code

        if bit index = 4096 * 8
            write bytes
            reset bit index
    repeat for sym

flush pairs(outfile)
    int bytes
    if bit index is not 0
        calculate bytes (divide by 8)
    else
        calculate bytes (divide by 8 + 1)
    write bytes

read pair(infile, code, sym, bit length)
    iterate from 0 to bit length
        if bit index is 0
            read bytes
        get bit and see if it = 1
            set bit
        else
            clear bit
        increment bit index

        if bit index = 4096 * 8
            reset bit index
    repeat for sym

buffer word(outfile, word)
    total bits = 0
    loop from 0 to length
        increment sym buffer index and set = to w->syms[i]
        if sym index = 4096
            write bytes
            reset sym index

    calculate total bits

flush words(outfile)
    if sym index is not 0
        write bytes
        reset sym index
```

Design Process

Over the course of this lab, I hardly modified my design since my overall understanding of the what the program is doing matched the pseudocode I have, of course with the help of the TA's (specifically Eugene and Oly). However, I did have several little issues that caused me to have to debug for hours straight.

- I modified how I got, set, and cleared my bits
- Removed some error checking
- Altered variable placements

All in all, the design of this lab was not too difficult due to spending multiple hours in lab sections thoroughly understanding the specifications for this assignment.