Trie-l and Error

Angela Shen

March 2023

1 Introduction

LZ78 is implemented by creating a trie, which is a tree that represents each letter in a message as a node and each word as a branch on the tree. The codes from this tree and the characters each node represent are paired together and outputted into a compressed file. This file can be decompressed using a word table. By reading each pair and adding the symbols to words on the word table, the file can be decompressed to derive the original message.

2 Trie

Implements the functions necessary to construct a Trie.

TrieNode *trie_node_create(uint16_t index): Constructor for non-root trie nodes. Set code equal to index and children to NULL.

```
node = dynamically allocate node
node code = index
for i in range (0, 256)
node children = NULL
```

void trie_node_delete(TrieNode *n): Destructor for a single node. Frees memory to the node pointer and sets pointer equal to NULL.

```
free(n)
n = NULL
```

TrieNode *trie_create(void): Root trie constructor. Dynamically allocates memory for TrieNode, sets code equal to the macro EMPTY_CODE, sets children to NODE.

```
node = dynamically allocate node
node code = EMPTY_CODE
for i in range (0, 256)
node children = NULL
```

void trie_reset(TrieNode *root): Set a trie to just the root node and recursively delete the children.

```
for i in range(0, 256)
    if (child == NULL)
        continue
    trie_delete(node children[i])
    node children[i] = NULL
```

void trie_delete(TrieNode *n): Deletes the current node and recursively calls itself to delete the child nodes.

```
for i in range(0, alphabet length)
    if (children[i] == NULL)
        continue
    trie_delete(Trienode *children[i])
    node children[i] = NULL
trie_node_delete(n)
n = NULL
```

TrieNode *trie_step(TrieNode *n, uint8_t sym): Searches the tree for the specified symbol. Otherwise, return NULL.

```
return node children[sym]
```

3 Word

Implements the functions necessary to create individual Words and a WordTable.

Word *word_create(uint8_t *syms, uint32_t len): Constructor for word. Dynamically allocate an array of uint8_ts (syms).

```
w = dynamically allocate a word
w syms = dynamically allocate an array of symbols
w len = len
for i in range(1, len)
    w->syms[i] = syms[i]
return w
```

Word *word_append_sym(Word *w, uint8_t sym): Creats a new word from w with sym appended to syms.

```
new = dynamically allocate a Word
new syms = dynamically allocate an array of symbols
for i in range(0, w->len)
        new->syms[i] = w->sym[i]
sym[w len] = sym
new->len = w->len + 1
return new
```

Deletes a word and frees memory. void word_delete(Word *w):

```
free(syms)
free(w)
```

WordTable *wt_create(void): Makes a word table, and array of Words of length MAX_CODE. Sets the first Word to the empty word.

```
dynamically allocate WordTable
empty = word_create(0, 0)
wt[0] = empty
return wt
```

void wt_reset(WordTable *wt): Resets Wordtable, leaving just the empty word.

```
for i in range(2, MAX_CODE)
    if wt[i] != NULL
        word_delete(wt[i])
        wt[i] = NULL
```

void wt_delete(WordTable *wt): No description, presumably free the entire WordTable.

```
for i in range(1, MAX_CODE)
    if wt[i] != NULL
```

```
        word_delete(wt[i])
        wt[i] = NULL
free(wt)
wt = NULL
```

4 IO

Functions to read symbols/pairs from infiles and output symbols/pairs to outfiles.

int read_bytes(int infile, uint8_t *buf, int to_read): Loops calls to read() until no more bytes can be read.

```
bytes_read = 0
x = 0
while(bytes_read != to_read)
    x = read(infile, buf, (to_read - bytes_read))
    if (x < 1)
        break
    bytes_read += x
return bytes_read
```

int write_bytes(int outfile, uint8_t *buf, int to_write): Loops calls to write() until no more bytes can be written.

```
bytes_write = 0
x = 0
while(bytes_write != to_write)
    x = write(outfile, buf, (to_write - bytes_write))
    if (x < 1)
        break
    bytes_write += x
return bytes_write
```

void read_header(int infile, FileHeader *header): Reads the header from infile and verifies the magic number. Swaps endianness if on a big endian machine.

```
read(infile, header, sizeof(FileHeader))
if (big_endian())
    header->magic = swap32(header->magic)
assert(header->magic == 0xBAADBAAC)
```

void write_header(int outfile, FileHeader *header): Write the header from outfile and verifies the magic number. Swaps endianness if on a big endian machine.

```
if (big_endian())
    header->magic = swap32(header->magic)
write(outfile, header, sizeof(FileHeader))
```

bool read_sym(int infile, uint8_t *sym): Reads symbols from the infile of encode and places them into the buffer.

```
x = 0
if (byte_left == 0)
    for i in range(0, BLOCK)
        buf_char[i] = 0
    buf_index = 0
    x = read_bytes(infile, buf_char, sizeof(buf_char))
    if (x < 1)
```

```
        return false
    byte_left += x
if (byte_left > 0)
    sym = buf_char[buf_index]
    total_syms++
    buf_index++
    byte_left--
return byte_left > 0
```

void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen): Writes pairs to the outfile of encode. This is done through bitwise operations (setting bitlen amount of bits and then another 8 bits for sym).

```
for i in range (0, bitlen)
    buf_pair[pair_index / 8] |= ((code >> i) & 1) << (pair_index % 8)
    pair_index++
    total_bits++
    if (pair_index == (BLOCK * 8))
for i in range(0, 8)
    buf_pair[pair_index / 8] |= ((sym >> i) & 1) << (pair_index % 8)
    total_bits++
    if (pair_index == (BLOCK * 8))
        flush_pairs(outfile)
```

void flush_pairs(int outfile): Flushes any pairs in the buffer into the outfile of encode.

```
write_bytes(outfile, buf_pair, (pair_index / 8))
for i in range (0, BLOCK, i++)
    buf_pair[i] = 0
pair_index = 0
```

bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen): Reads pairs from the infile of decode into the buffer.

```
code = 0
sym = 0
uint32_t x = 0
if (bits_left == 0)
    x = reset_buffer(infile)
    if (x < 1)
        return false
if (bits_left > 0)
    for i in range(0, bitlen)
        code |= ((buf_pair[pair_index / 8] >> (pair_index % 8)) & 1) << i
        pair_index++
        total_bits++
        bits_left--
        if (pair_index == (BLOCK * 8))
            reset_buffer(infile)
    for i in range(0, 8)
        sym |= ((buf_pair[pair_index / 8] >> (pair_index % 8)) & 1) << i
        pair_index++
        total_bits++
        bits_left--
        if (pair_index == (BLOCK * 8))
            reset_buffer(infile)
```

```
if (code == STOP_CODE)
    return false
return (bits_left > 0)
```

uint32_t reset_buffer(int infile): Helper function for read_pair, sets the buffer to 0 if the buffer is empty or full.

```
x = 0
for i in range (0, BLOCK)
    buf_pair[i] = 0
pair_index = 0
x = read_bytes(infile, buf_pair, sizeof(buf_pair))
if x < 1
    return x
bits_left += (x * 8)
return x
```

void write_word(int outfile, Word *w): Writes words from the wordtable into the outfile of decode.

```
for i in range(0, w->len)
    buf_char[buf_index] = w->syms[i]
    buf_index++
    total_syms++
    if (buf_index == BLOCK)
        flush_words
```

void flush_words(int outfile): Flushes any words in the buffer to the outfile of decode.

```
write_bytes(outfile, buf_char, buf_index)
for i in range (i < BLOCK)
    buf_char[i] = 0
buf_index = 0
```