CSE 13S: Assignment 6 Design Doc

Johnny Li

March 3, 2023

1 Introduction

In recent years, both data compression and data security have emerged to the forefront of software engineering. These two fields have had a massive impact on the modern computing landscape, playing a vital role by underpinning many technologies we take for granted. From mobile data, internet of things (IoT), cloud computing and beyond, data compression and security have become essential components of our digital lives.

Data compression algorithms reduce the number of bits required to represent data, resulting in significantly smaller data size which allows for faster transfers and reduced storage costs. There are two types of compression algorithms, lossy and loseless. Lossy compression algorithms compress data more efficiently than lossless algorithms, since they choose to throw away information; that data cannot be recovered. Lossy compression is often used for audio and video files which can lose a certain amount of data without impacting the listening or viewing experience. Lossless compression algorithms, on the other hand, do not lose any information; they are usually used to compress files which must maintain their integrity such as text documents, binaries, and source code.

A good example of lossy versus loseless compression can be given with YouTube videos which are compressed with lossy compression. When there is a consistent, relatively still portion of the video, the quality and frame rate appears normal but when something such as snowfall or confetti appears on the screen, the compression algorithm will struggle to keep up and the quality of the video will decrease. This is because the lossy compression algorithm discards some of the data in the video, resulting in a loss of quality when there is a sudden increase in complexity.

2 Lempel-Ziv Compression

Lempel-Ziv (LZ) compression is a family of lossless data compression algorithms that includes LZ77 and LZ78. The LZ77 algorithm, published in 1977 by Abraham Lempel and Jacob Ziv, is based on the idea of replacing repeated patterns

in data with pairs that consist of a code and a symbol. The code is an unsigned 16-bit integer, and the symbol is an 8-bit ASCII character.

The LZ78 algorithm, published in 1978, is a variant of LZ77 that uses a different approach to constructing the dictionary of patterns. Both algorithms work by building a dictionary of previously seen patterns, and then replacing repeated patterns with a pair consisting of the index of the pattern in the dictionary and the next symbol in the input stream. The compression ratio of LZ algorithms depends on the frequency of repeated patterns in the input data.

The LZ77 algorithm uses a sliding window to keep track of previously seen patterns. The window is a fixed-size buffer that contains the most recent part of the input stream. The algorithm scans the input stream, looking for patterns that match the contents of the window. When a match is found, the algorithm outputs a pair consisting of the index of the matching pattern in the window and the next symbol in the input stream. The algorithm then moves the window forward by the length of the matching pattern, so that it includes the next part of the input stream.

The LZ78 algorithm uses a different approach to building the dictionary of patterns. It starts with an empty dictionary, and then scans the input stream, looking for the longest prefix of the input that is already in the dictionary. When a new prefix is found, it is added to the dictionary, and a pair consisting of the index of the prefix in the dictionary and the next symbol in the input stream is output. The algorithm then continues scanning the input stream, adding new prefixes to the dictionary as they are found.

3 Task

Our task is to implement two programs called `encode` and `decode` which perform LZ78 compression and decompression, respectively. The requirements for the programs are as follows:

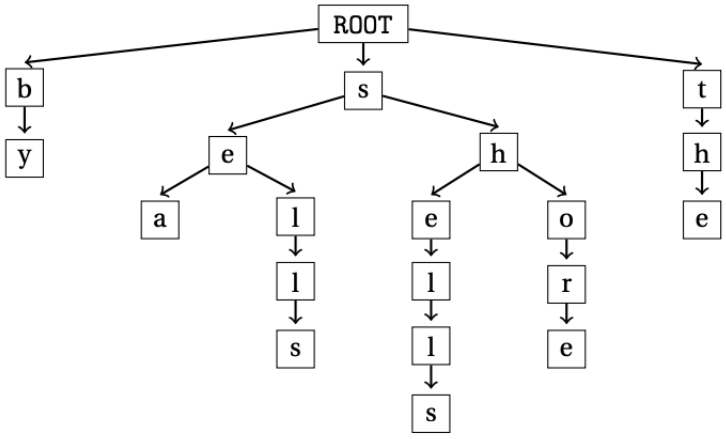
1. `encode` can compress any file, text or binary.
2. `decode` can decompress any file, text or binary, that was compressed with `encode`.
3. Both operate on both little and big endian systems. Interoperability is required.
4. Both use variable bit-length codes.
5. Both perform read and writes in efficient blocks of 4KB.
6. Both `encode` and `decode` must interoperate with the provided binaries – not just your code.

New abstract data types (ADTs) will need to be implemented for this assignment: an ADT for tries and an ADT for words. Additionally, we'll also need to

implement variable-length codes, I/O, and big-little endianness intercompatibly.

3.1 Tries

The most computationally challenging part of compression is checking for existing prefixes, or words. While this could be done with an array or hash table, that wouldn't be efficient. Instead, a trie could be used instead. An efficient information retrieval data structure, it's also known as a prefix tree. Each node in a trie represents a symbol or character and contains n child nodes, where n is the size of the alphabet you are using. In most cases, the alphabet used is the set of ASCII characters, so $n = 256$. Our implementation will use a trie during compression to store words.



The image above is a trie containing the following words: "She", "sells", "sea", "shells", "by", "the", "sea", "shore". Seaching through the trie involves going down the appropriate branch for each symbol in the word, starting from the root at the top. Moving down the trie involves verifying whether the current node has a child node representing the symbol being sought. If such a child node exists, the current node becomes that child node. To locate "sea", for example, one begins at the root of the trie and descends to 's', then 'e', and finally 'a'. If a symbol is absent or the end of the trie is reached without fully matching a word while traversing the trie, the word is not present in the trie.

TrieNode

```
struct TrieNode {
    TrieNode *children[ALPHABET];
    uint16_t code;
};
```

The `TrieNode` struct has two fields with each trie node having an array of 256 pointers to trie nodes as children, one for each ASCII character. The `code` field stores the 16-bit code for the word that ends with the trie node containing the code. This means that the code for some word "abc" would be contained in the trie node for 'c'. Note that there isn't a field that indicates what character a trie node represents. This is because the trie node's index in its parent's array of child nodes indicates what character it represents. The `trie_step()` function will be repeatedly called to check if a word exists in the trie. A word only exists if the trie node returned by the last step corresponding to the last character in the word isn't `NULL`.

```
TrieNode *trie_node_create(uint16_t index)
```

The constructor function for a `TrieNode` that sets the node's code to `code` and initializes each of its children node pointers to `NULL`.

```
void trie_node_delete(TrieNode *n)
```

The destructor function for a `TrieNode`; it takes a single pointer as input. Unlike the destructors from previous assignments, it does not require double pointers to NULL the pointer by dereferencing it. Using a single pointer makes the code more manageable and simplifies the overall implementation.

```
TrieNode *trie_create(void)
```

Initializes a trie by creating a root `TrieNode` with the code `EMPTY_CODE`. If successful, it returns a pointer to the root node (`TrieNode*`), otherwise it returns `NULL`.

```
void trie_reset(TrieNode *root)
```

Resets a trie to only have the root `TrieNode`. If the maximum available code, (`MAX_CODE`), is reached during compression/decompression, the trie must be reset by deleting all of its children nodes. To achieve this, each of the root node's children nodes must be set to `NULL`.

```
void trie_delete(TrieNode *n)
```

Deletes a sub-trie starting from the `TrieNode` rooted at node `n`. It requires recursive calls on each of `n`'s children nodes. After calling `trie_node_delete()` on each child, the pointer to the child node must be set to `NULL` to avoid any dangling pointers.

```
TrieNode *trie_step(TrieNode *n, uint8_t sym)
```

This function returns a pointer to the child node representing the symbol `sym`. If the symbol does not exist, it returns `NULL` instead.

3.2 Word Tables

To facilitate quick code-to-word translation during decompression, a new struct called `WordTable` will be defined. Since there is a maximum of 2^16-1 codes, with one reserved as a stop code, a fixed word table size of `UINT16_MAX` as defined in `inttypes.h`. `UINT16_MAX` as the maximum value of an unsigned 16-bit integer, and has the exact same value as `MAX_CODE`.

Hash tables will not be used to store words because there is a fixed number of codes. Instead, each index of this word table will be a new struct called a `Word`. Words will be stored in byte arrays, or arrays of `uint8_t`, because strings in C are null-terminated and problems with compression can occur if a binary file being compressed contains null characters that are placed into strings. Since the length of a word needs to be known, a `Word` will also have a field for storing the length of the byte array, since `string.h` functions like `strlen()` can't be used on byte arrays.

Word

```
typedef struct Word {
    uint8_t *syms;
    uint32_t len;
} Word;
```

A `Word` is a struct that holds an array of symbols called `syms`, which are stored as bytes in an array. The length of the array storing the symbols that a `Word` represents is stored in `len`.

```
typedef Word * WordTable
```

An array of `Word`s will be defined as a `WordTable`.

```
Word *word_create(uint8_t *syms, uint32_t len)
```

Constructor for a `Word` where `syms` is the array of symbols a `Word` represents. The length of the array of symbols is given by `len`. This function returns a `Word *` if successful or `NULL` otherwise.

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

Constructs a new `Word` from the specified `Word`, `w`, appended with a symbol, `sym`. If the `Word` specified to append to is empty, the new `Word` should contain only the symbol. The function returns the new `Word`, which represents the result of appending.

```
void word_delete(Word *w)
```

A destructor function for a `Word`, `w`. Like with `trie_node_create()`, a single pointer is used here to reduce the complexity of memory management, thus reducing the chances of having memory-related errors.

```
WordTable *wt_create(void)
```

Creates a new WordTable, which is an array of Words. A WordTable has a predefined size of |MAX_CODE—, which has the value |UINT16_MAX—. This is because codes are 16-bit integers. A WordTable is initialized with a single Word at index |EMPTY_CODE—. This Word represents the empty word, a string of length of zero.

```
void wt_reset(WordTable *wt)
```

Resets a WordTable, `wt`, to contain just the empty Word. All the other words in the table should be `NULL`.

```
void wt_delete(WordTable *wt)
```

A destructor function for `wt` `wt`. Like with both `trie_node_create()` and `void word_delete(Word *w)`, a single pointer is used here to reduce the complexity of memory management, thus reducing the chances of having memory-related errors.

3.3 Codes

When encoding a pair with a variable bit-length code, it is necessary to determine the minimum number of bits needed to represent the code. This can be calculated using the formula $b = log_2(x)c + 1$, where x is the integer to be represented and c is the number of bits needed to represent the next available code.

For example, if the pair $(13,' a')$ is being encoded and the next available code is 64, the bit-length of the code is 7. Starting from the least significant bit of 13, the code portion of the pair's binary representation is constructed, with padded zeroes to fill in the remaining bits of the code. Similarly, the symbol portion of the pair's binary representation is constructed by starting from the least significant bit of 'a' and adding it to the code portion.

When decoding a binary representation of a pair, the bit-length of the code is determined by the bit-length of the next available code assigned during compression. The bits representing the code are then summed using positional numbering, and the resulting integer is the code of the pair. The same process is repeated for the symbol portion of the binary representation.

```
#ifndef __CODE_H__
#define __CODE_H__

#include <inttypes.h>

#define STOP_CODE 0
#define EMPTY_CODE 1
```

```
#define START_CODE 2
#define MAX_CODE UINT16_MAX

#endif
```

3.4 I/O

When implementing the programs encode and decode, efficient input/output (I/O) is necessary. In order to achieve this, reads and writes will be done in blocks of 4KB, which requires buffering of I/O. Buffering is the process of storing data into a buffer, which can be considered as an array of bytes. An I/O module is required for this assignment, with an API that is explained in the subsequent sections.

```
FileHeader
typedef struct FileHeader {
    uint32_t magic;
    uint16_t protection;
} FileHeader;
```

Defines a struct for the file header, which includes the magic number and protection bit mask of the original file. The magic number, represented by the "magic" field, serves as a unique identifier for compressed files created by the "encode" program. The value of the magic number is 0xBAADBAAC.

Prior to writing the file header to the compressed file using the "write_header()" function, the endianness of the fields must be swapped if necessary for interoperability. If the program is running on a big-endian system, the fields should be swapped to little endian, which is canonical. An endianness module will be provided for this purpose.

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

A helper function that reads a specified number of bytes from a file descriptor `fd` and stores them in a buffer `buf`. It loops over calls to the `read()` system call until it has read all the bytes specified by `to_read` or there are no more bytes to read before returning the number of bytes that were read.

```
int write_bytes(int outfile, uint8_t *buf, int to_write)
```

This function is similar to `read_bytes()`, but it is used for looping calls to `write()`. Since `write()` is not guaranteed to write out all the specified bytes (to_write), we need to loop until we have either written out all the bytes specified, or no bytes were written. The function returns the number of bytes written out. It should be used whenever a write operation is performed.

```
void read_header(int infile, FileHeader *header)
```

This reads in `sizeof(FileHeader)` bytes from the input file. These bytes are read into the supplied header. Endianness is swapped if byte order isn't little endian. Along with reading the header, it must verify the magic number.

```
void write_header(int outfile, FileHeader *header)
```

Writes `sizeof(FileHeader)` bytes to the output file. These bytes are from the supplied header. Endianness is swapped if byte order isn't little endian.

```
bool read_sym(int infile, uint8_t *sym)
```

An index keeps track of the currently read symbol in the buffer. Once all symbols are processed, another block is read. If less than a block is read, the end of the buffer is updated. Returns `true` if there are symbols to be read, `false` otherwise.

```
void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)
```

`Writes` a pair to `outfile`. In reality, the pair is buffered. A pair is comprised of a code and a symbol. The bits of the code are buffered first, starting from the LSB. The bits of the symbol are buffered next, also starting from the LSB. The code buffered has a bit-length of `bitlen`. The buffer is written out whenever it is filled.

```
void flush_pairs(int outfile)
```

`Writes out` any remaining pairs of symbols and codes to the output file.

```
bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)
```

`Reads` a pair (code and symbol) from the input file. The `read` code is placed in the pointer to code (e.g. `*code = val`). The `read` symbol is placed in the pointer to sym (e.g. `*sym = val`). In reality, a block of pairs is read into a buffer. An index keeps track of the current bit in the buffer. Once all bits have been processed, another block is read. The first `bitlen` bits are the code, starting from the LSB. The last 8 bits of the pair are the symbol, starting from the LSB. Returns `true` if there are pairs left to read in the buffer, else `false`. There are pairs left to read if the read code is not `STOP_CODE`.

```
void write_word(int outfile, Word *w)
```

Writes a pair to the output file. Each symbol of the Word is placed into a buffer. The buffer is written out when it is filled.

```
void flush_words(int outfile)
```

Writes any remaining symbols in the buffer to the output file. It is important to note that the output file must have the same protection bits as the original file. The function utilizes the `fstat()` and `fchmod()` functions, similar to Assignment 4. All reads and writes within the program are performed using the

system calls `read()` and `write()`. Therefore, the function uses the system calls `open()` and `close()` to obtain the file descriptors. It is necessary to perform all reads and writes in efficient blocks of 4KB. To achieve this, two static 4KB `uint8_t` arrays serve as buffers: one to store binary pairs and the other to store characters. Each of these buffers should have an index or variable to keep track of the current byte or bit that has been processed.

3.5 Compression Pseudocode

```
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
        WRITE_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
    WRITE_PAIR(outfile, prev_node.code, prev_sym, BIT_LENGTH(next_code))
    next_code = (next_code +1) % MAX_CODE
WRITE_PAIR(outfile, STOP_CODE, 0, BIT_LENGTH(next_code))
FLUSH_PAIRS(outfile)
```

3.6 Decompression Psuedocode

```
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)
    WRITE_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)
```

3.7 Program Options

The encode program will support the following commmand-line options:

- `-v`: Print compression statistics to stderr.
- `-i <input>`: Specify input to compress (stdin by default).
- `-o <output>`: Specify output of compressed input (stdout by default).

The decode program will support the following commmand-line options:

- `-v`: Print decompression statistics to stderr.
- `-i <input>`: Specify input to decompress (stdin by default).
- `-o <output>`: Specify output of decompressed input (stdout by default).

The `-v` option enables verbose to print out informative statistics about the compression or decompression that is performed. These statistics include information about the compressed file size, the uncompressed file size, and space saving. The formula for which is: $100 \times \left(1 - \frac{\text{compressed size}}{\text{uncompressed size}}\right)$

The verbose output will be structured like this:
Compressed file size: X bytes
Uncompressed file size: X bytes
Space saving: XX.XX%

4 Sources

- Psuedocode provided by TA John Yu
- The C Programming Language