Assignment 6 DESIGN Document

Sam Leger

March 12, 2023

Professor Long
CSE13S Winter Quarter

1 Description of Program

Compression is the act of reducing the number of bits required to represent data. This technique is invaluable, allowing for increased speed and capacity when transferring data. There are two types of data compression algorithms- lossy and lossless. The former compresses more data, with the downside of losing some of it. For this reason, it is used primarily when dealing with audio and video, where drops in quality are less important. Lossless compression is used when the data must remain intact, like when dealing with text files, binary programs, or code. This program contains the implementation of the Lempel-Ziv (LZ78), a lossless compression algorithm that was devised in the late seventies. This program contains both an encoder (compresses files), and a decoder (decompresses files). These operate on both little-endian and big-endian systems.

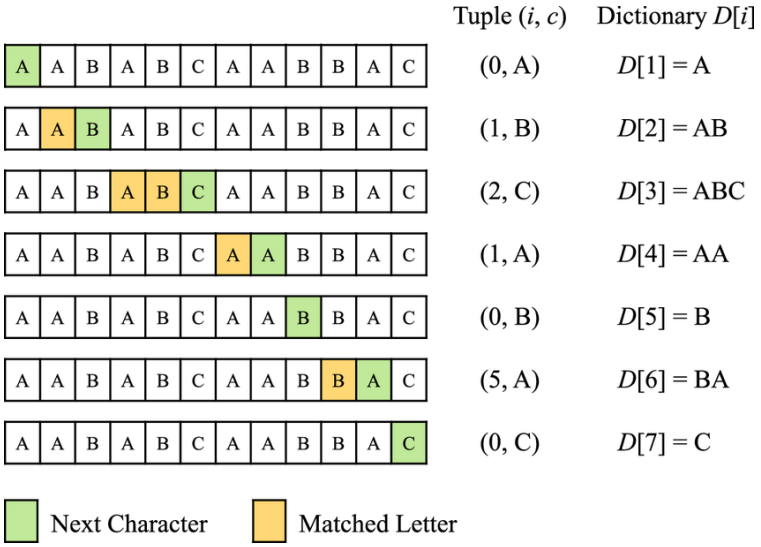

Figure 1: LZ78 Process

2 Tries

A trie, also known as a prefix tree, is used to store and search for a specific key from a set. In this case, it is used to store words. Each node represents a symbol, and each of its children represents a symbol that might come after in a word. For example, say we have the following words: *cat*, *car*, and *cart*. Our trie would look like this:

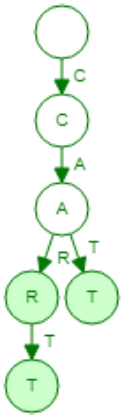

Figure 2: Example Trie

In this program, a trie is an abstract data type (ADT) that contains an array of 256 (one for each ASCII character) pointers to trie nodes as children. The struct also contains an unsigned 16-bit code that is used for the corresponding code. Here is the implementation of the `TrieNode struct`:

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

six different functions have been implemented to work with tries and trie nodes.

- `*trie_node_create()`

```
TrieNode *trie_node_create(uint16_t code) {
    - allocate memory for TrieNode object
    - set node->code = code
    - loop through all possible children
        - set each child to NULL
    - return new node object
}
```

- trie_node_delete()

```
void trie_node_delete(TrieNode *n) {
    - if node pointer is not NULL
        - loop through all possible children
            - recursively call trie_node_delete() on child
        - free node pointer
}
```

- *trie_create()

```
TrieNode *trie_create(void) {
    - create a new TrieNode object with EMPTY_CODE
    - if root is not NULL
      - return root
    - else
      - return NULL
}
```

- trie_reset()

```
void trie_reset(TrieNode *root) {
    - loop through all possible children
        - call trie_node_delete() on child
        - set child node pointer to NULL
}
```

- trie_delete()

```
void trie_delete(TrieNode *n) {
    - if TrieNode n is NULL
        - loop through all possible children
            - recursively call trie_delete() on child
            - set child node pointer to NULL
        - free TrieNode n
}
```

- *trie_step()

```
TrieNode *trie_step(TrieNode *n, uint8_t sym) {
    -if TrieNode n is not NULL
        - return child node corresponding to sym
    - else
        - return NULL
}
```

3 Word Tables

The second ADT used in this program is a word table, which is used in decompression. A word table is used as a look-up for quick code-to-word translation. A word table is an array of words, which contain symbols stored as bytes in an array. The length of the array is stored in a second variable called len. The length of the array is necessary because functions like strlen() cannot be used on byte arrays. Here is the implementation of the word `struct`:

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

As seen above, `syms` is a pointer to an array of unsigned 8-bit integers, representing a sequence of symbols. `len` is an unsigned 32-bit integer that represents the number of elements in `syms`. The word module contains all functions related to words and word tables. Here are the other functions for the manipulation of words and word tables:

- `word_create()`

```
Word *word_create(uint8_t *syms, uint32_t len) {
    - allocate memory for word object and syms array
    - loop through syms and copy them into the new array
    - set w->syms to new array
    - set w->len to len argument
    - return new word object
}
```

- `word_append_sym()`

```
Word *word_append_sym(Word *w, uint8_t sym) {
    - calculate new len = len + 1
    - allocate memory for new syms array
    - loop through syms and copy them into the new array
```

```
        - append the new symbol to the end of the array
        - call word_create(new_syms, new_len)
        - return new word object
    }
```

- word_delete()

```
    void word_delete(Word *w) {
        - free w->syms
        - free w
    }
```

- *wt_create()

```
    WordTable *wt_create(void) {
        - allocate memory for the WordTable of MAX_CODE length
        - create the first (EMPTY) word in the table
        - return word table object
    }
```

- wt_reset()

```
    void wt_reset(WordTable *wt) {
        - iterate through word table
            - if word is not NULL
                - set word to NULL
    }
```

- wt_delete()

```
    void wt_delete(WordTable *wt) {
        - iterate through word table
            - if word is not NULL
                - call word_delete() on it
        - free word table object
    }
```

4 Codes

The LZ78 program is able to work by reading and writing bit-length codes. First, the minimum number of bits required to represent a number is calculated using the following formula: $\log_2(x)+1$. Once the size is found, the size of the number that is going to be used is found. If the numbers are different, zeroes are padded to the end based on where the most significant byte (MSB) is. This depends on the endianness.

5 I/O

Since the encode and decode functions need to perform efficient I/O, a module is implemented to deal with input and output. The reads and writes are done in 4KB blocks, which requires the use of a buffer. This struct contains an unsigned 32-bit magic number, which serves as a unique identifier for files that have been compressed with the encoder.

- `read_bytes()`

```
        - loop calls to read() until
            - all bytes are read
            - no more bytes to read
```

- `write_bytes()`

```
        - loop calls to write() until
            - all bytes are written
            - no more bytes to write
```

- `read_header()`

```
        - reads header bytes from the input file
        - swaps endianness if necessary
        - verify magic number
```

- `write_header()`

```
        - writes header to the output file
        - swaps endianness if necessary
```

- `read_sym()`

```
        - processes symbols in blocks
        - if less than a block is read, updates the buffer
        - returns true if symbols are left
        - false otherwise
```

- `write_pair()`

```
        - writes a pair to outfile
        - pairs are comprised of a code and a symbol
        - the code is buffered first, then the symbol
```

- `flush_pairs()`

```
        - write out remaining pairs to outfile
```

- `read_pair()`

```
        - reads a pair from the input file
        - pairs are comprised of a code and a symbol
        - the code is read first, then the symbol
        - check if there are pairs left using STOP_CODE
```

- `write_word()`

```
        - writes a pair to the output file
```

- `flush_words()`

```
        - writes out any remaining symbols to the output file
```

6 Encode

The encoder starts by parsing command line options. The user can specify the input file (to be compressed) and the output file (to send the compressed file). After options are handled, a header is created and a magic number and protection bits are assigned. The permissions are then changed to that of the input file. `write_header()` is called to write out the header, and then the compression process occurs. After all pairs are flushed, there is a check if verbose output is enabled for display. Lastly, input and output files are closed using `close()`.

7 Decode

Like the previous program, the decoder starts by looping through command line options. The user can specify the input file (to be decompressed) and the output file (to send the decompressed file). After, a header is created and populated using `write_header()`. The header is then verified to have the correct magic number, and permissions for the file are changed. Decompression occurs, and the check for verbose output follows. Just like the encoder, input and output files are closed using `close()`.

8 Credit and Resources

The two biggest resources used in this assignment were the pseudocode in the asgn6 document provided by Professor Long, and the function explanations from the TAs included in the header files. I also made sure to utilize the test cases provided in the resources directory, and they were useful in making sure my functions met some of the conditions. Other resources used were the class slides, specifically 18 - *Linked Lists*, and 20 - *Data Compression*.