CSE13S Assignment 6 Design

Zion Kang

February 27 2023

1 Description

In this assignment, we implement two programs called encode and decode which perform LZ78 compression and decompression, respectively. The programs utilize two different ADTs during these processes: tries and word tables. In order to accomplish LZ78 compression and decompression, we also implement a codes program that reads and writes pairs containing variable bit-length codes as well as a program that performs efficient I/O.

2 Pseudocode

Tries:

trie_node_create:
Constructor for a TrieNode. The node's code is set to the passed in code.
Allocate memory for the TrieNode ADT.
Each children node should point to NULL.

trie_node_delete:
Destructor for a TrieNode where only a single pointer is passed in.
Free the memory allocated for the node.
(Setting the pointer to NULL is done later in trie_delete where you recurisvely call on itself as you free memory for each TrieNode and set each points to NULL.)

trie_create:
Initialize a trie: a root TrieNode with the code EMPTY_CODE.
Return the root, a TrieNode *, if successful, NULL otherwise.

trie_reset:
Reset a trie to just the root TrieNode.
Since we have a finite number of codes, we will eventually arrive at MAX_CODE.
When we reach this point, reset the trie by deleting its children and setting each pointer to NULL so that we can continue compressing/decompressing the file.
This function essentially just loops through the trie and deletes all the children nodes and sets their pointers to NULL so we can continue to compress/decompress more data.

trie_delete:
Deletes a sub-trie starting from the trie rooted at node n.
In order to do this, recursively call on each of n's children (TrieNodes).
As we make recursive calls on each of n's TrieNode children, set each pointer to NULL after we free them with the trie node delete function.

trie_step:

Looping through a trie, we return the pointer to the child node representing the symbol sym.
If the symbol doesn't exist within the trie, return NULL.

Word Tables:

word_create:
Constructor for a word where syms is the array of symbols a Word represents.
Allocate memory for the Word ADT.
The length of this array of symbols (or word) is given by len.
This function should return a Word * if successful or NULL otherwise.

word_append_sym:
Constructs a new Word from the specified Word, w, appending with a symbol, sym.
The Word appended may be empty.
If Word is empty, the new Word should contain only the symbol.
Return the new Word which represents the result of appending

word_delete:
Destructor for a Word, w.
Free the word and set its pointer to NULL.

wt_create:
Creates a WordTable, an array of Words.
Initialize a WordTable with a single Word at index EMPTY_CODE. This Word represents the empty word, a string of length of zero in our dictionary.

wt_reset:
Resets a WordTable, wt, to contain just the empty Word.
Traverse the WordTable and delete all words except for obviously the empty word at the beginning.

I/O:

read_bytes:
This is a helper function to perform reads.
Because read() syscall doesn't always read all the bytes specified, we write this function to read until we have read all the specified bytes or there are no more bytes to read.
The number of bytes that were read are returned.

write_bytes:
This function is basically the same as read_bytes(), except that we loop calls to write().
Once again, write() isn't guaranteed to write out all the specified bytes so this function loops until we have actually written out all the bytes specified, or no bytes are to be written.
The number of bytes written is returned.

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

write_header:
Writes the size of the header file (specified by passed in header) to the output file.
Once again, endianness is swapped if byte order isn't little endian.

read_sym:
Keeping track of the currently read symbol in the buffer, read in all symbols.
Once all symbols are processed, another block is read.
If less than a block is read, the end of the buffer is updated to the specified read bytes.
Return true if there are symbols to be read, false otherwise.

write_pair:
Write a pair to the outfile (buffer).
A pair is comprised of a code and symbol.
The bits for the code are buffered first, starting from LSB.
The bits of the symbol are buffered next, also starting from the LSB.
The code buffered has a bit length of bitlen and the buffer is written out whenever it is filled.

flush_pairs:
Writes out any remaining pairs of symbols and codes to the output file, so as to "flush out" the pairs.
reset the memory in the buffer to 0
reset index in buffer to 0

read_pair:
"Reads" a pair (code and symbol) from the input file into the pointer to code.
The read symbol is placed in the pointer to sym.
This function is used to read in a block of pairs into a buffer.
An index keeps track of the current bit in the buffer and once all bits have been processed, another block is read.
The first bitlen bits are the code, starting from LSB. The last 8 bits of the pairs are the symbol, starting from LSB.
Return true if there are pairs left to read in the buffer, else false. (Note: There are pairs left to read if the read code is not STOP_CODE).

read_word:
"Writes" a pair to output file where each symbol of the Word is placed into a buffer.
The buffer is written out when it is filled.

flush_words:
Write out any remaining symbols in the buffer to the outfile.
reset memory in buffer to 0
reset buffer index to 0

Compression:

1. Open infile with open(). If an error occurs, print a error message and exit with error code. (Default: stdin)

2. The first thing in the infile must be the file header and it's magic number must be 0xBAADBAAC. Also check for the file size and protection bit mask here.

3. Open the outfile and check that the permissions for it match the protection bits set in the file header. If an error occurs while trying to open the outfile, print an error message and exit with error code. (Default: stdout)

4. Write the filled out file header to outfile. Note: This means writing out the struct itself to the file.
5. Create a trie. This trie should have no children and only the root.
The code stored by this root should be EMPTY_CODE to denote the empty word.
After creating a copy of the root node, use it to step through the trie to check for existing prefixes.
This root node copy will be referred to as curr_node and the reason a copy is needed is that we will eventually need

to reset whatever trie node we've stepped to back to the top of the trie. Using a copy lets us return back to the top.

6. We will need a monotonic counter to keep track of the next available code.
This counter should start at START_CODE and the counter should be uint16_t since the codes used are unsigned 16-bit integers. This will be referred to as next_code.

7. You will also need two variables to keep track of the previous trie and previously read symbol.
We will refer to these as prev_node and prev_sym.

8. Reading in all the symbols from infile and stopping when there are no more symbols to be read:
- call each read in symbol curr_sym
- set next_node to be the return on calling trie step with curr node and curr sym we're looking at, so as to step down from the current node to the current read symbol.
- if the next node isn't NULL, that means we've seen the prefix. update prev_node to be curr_node and curr_node to be next_node.
- else, we have not encountered the current prefix (next_node is NULL) and we write the pair out and add the current prefix to the trie. To do this, we let the current node's child be a new trie node whose code is next_code.
After this, reset where we look to the top of the trie and increment he value of the next_code so as to get the next unused code.
- Check if the next code is equal to max code and if it is, reset the trie to just having the root node.
- update the prev symbol to be the current symbol

9. After processing all the characters in infile, check if we're looking at the root trie node.
- If we are, it means we were still matching a prefix, so write the pair for the prev code/sym. After doing this, increment next_code and make sure it is within the limit MAX_CODE using modulo.

10. Once we have processed all characters in infile, write the pair for stop code to signal the end of the compressed output.

11. Call on flush pairs to flush any left over buffered pairs.

12. close infile and outfile

Decompression:

1. Open infile, if an error occurs print an error message with error exit code (Default: stdin).

2. Read in file header. If magic number is verified, then we can go ahead with decompression.

3. Open outfile. Permissions for this file should match the protection bits as set in read file header. If an error occurs when opening outfile, handle same as infile (Default: stdout).

4. Create word table and make sure each entry is NULL.
Initialize table to have just empty word at index EMPTY_CODE.

5. Utilize two uint16_ts to keep track of current code and next code.
next_code is to be initialized as START_CODE which serves as a monotonic counter.

6. Loop reading pairs in from infile until we reach STOP CODE and for each pair perform:
- We need to append the read symbol with the word denoted by the read code and add the result to table at the index next code.
- Write the word that we just created and added to the table.

- Increment the next unused code and check if it equals MAX CODE. If it is, reset the table and set next code to START CODE (similar to that of resetting a trie during compression).

7. Flush buffered words that have been buffered under the hood by the write functions.

8. close infile and outfile