Assignment 6 Writeup: Stuffing 0s and 1s Down the Filestream

Sep Nasiriany

Due Date: March 12, 2023

1 Introduction

The modern world relies on file compression in order to make best use of the finite amount of data we're given. In this assignment, I used Lempel-Ziv Compression to take a file and compress it using tries and nodes to avoid reusing space for the same characters/symbols. The Lempel-Ziv Compression relies on a symbol and code in a pair to get written in a compressed file, and saves space but reusing a shorter code (albeit up to 16 bits) to express a series of letters or a word. The application of compression is very important because many systems rely on compression in order to operate, especially older ones that had less memory to spare. The cloud that syncs our local data to huge servers still rely on compression to make sure that they are not overwhelmed with data. The internet itself heavily relies on compression because videos would load too slowly or files would load too slowly if they were to be displayed in their native format. The diagram below shows a very basic understanding of the compression process.

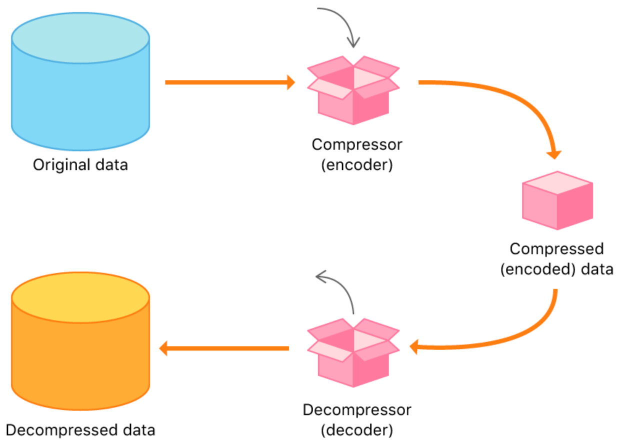


Figure 1
This image from Apple shows how basic compression requires a compression encoder which uses a specific algorithm and a decoder which reverses the process using a decoder.
`https://developer.apple.com/documentation/compression`

2 Lessons I learned:

The idea of Lempel-Ziv Compression made sense to me pretty quickly, however I found it quite hard to implement since I did not have any experience with more intricate file I/O. First, I learned quite a lot about tries and nodes on a pretty abstract level. I understood that nodes can have as many children as that data type can support. So if you were to make a node with a data type `uint8_t`, it can support up to 256 children. This is exactly how many pointers we needed for each node because it needed to represent all ASCII characters.

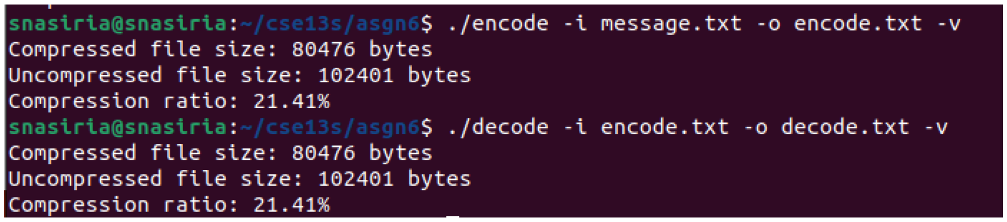
Memory allocation was also pretty big in this assignment. In the creation of `word.c` and `trie.c`, I had to allocate memory using `calloc` and free them accordingly using `free()`. While this process wasn't particularly challenging, it did require diligence in tracking what was being allocated and when it was freed. This also bolstered my understanding of data types since `Word` and `TrieNodes` were data types that had to be allocated and freed. Memory allocation was very crucial as it always is in C because if one thing went wrong, it can cause the entire program to crash and report a segmentation fault. One thing I also realized with regards to memory is that when arrays were statically made, their values were not set to 0 automatically. This is something you have to manually do and that means if you don't set set an index to a certain value, but use it later, it will report a random number that can cause all kinds of bugs. This specifically was the source of one of my bugs and caused 4 or 5 hours of debugging pain for me to figure out. The reason why C doesn't do this is because it does not want to waste any processing power to initalize the array for us and thus work faster.

I also learned quite a lot about manipulating bits and data. If this assignment doesn't make you understand how data can be represented in bits, decimals, hex, etc. then nothing will. I spent probably 10-20 hours just staring at `xxd` in hex and binary form and writing down on my notebook how all these bits would be affected. The fact that data was written from least significant bit (LSB) to most significant bit (MSB) meant that data had to be handled in a specific way not to disrupt that flow of order. I decided to just reverse its order back to MSB to LSB, do whatever bitwise manipulation I needed, and flip it back to LSB to MSB. This was one of things that's easier said than done because you have to make sure it's done correctly at all points or the whole encoded file is bogus. Finally, tracking bits within a byte proved to be challenging since there are a lot of possibilities to consider with regards to where the index position can be. Here is just 1 out of 15 pages of notes I wrote to understand how to do.

The last thing I learned was buffering data in file I/O and its purpose. In this assignment, we had to use `open()`, `read()`, and `write()` which are system calls that don't involve outside libraries. These calls are quite expensive and slow when you do them thousands or millions of times for 1 byte. In this assignment, we used a 4KB block array in order to use it as an intermediary between what we wanted to read/write and the file we wanted to access. This also meant that the buffer would need to get refilled if it was reading or writing more than 4KB. I found that even when I would compress massive files, everything in the program worked extremely quickly and instantly which I attribute to buffering data.

3 Efficiency

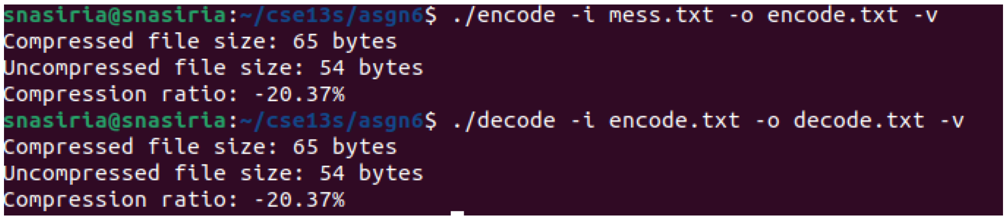
The question that ultimately challenges the Lempel-Ziv algorithm isn't really whether or not it works, because it does. The question is how much storage it saves without using a lot of processing power. This is ultimately what computer scientists assess when they want to choose compression algorithms. I decided to insert a 100LB file (seemed to be the limit before the linux text editor would crash) and saved a little over 20% in space. Here are the verbose details:



```
snasiria@snasiria:~/cse13s/asgn6$ ./encode -i message.txt -o encode.txt -v
Compressed file size: 80476 bytes
Uncompressed file size: 102401 bytes
Compression ratio: 21.41%
snasiria@snasiria:~/cse13s/asgn6$ ./decode -i encode.txt -o decode.txt -v
Compressed file size: 80476 bytes
Uncompressed file size: 102401 bytes
Compression ratio: 21.41%
```

Figure 2
This image demonstrates statistics of my Lempel-Ziv program with a 100KB file of text randomly generated by: `https://onlinefiletools.com/generate-random-text-file`

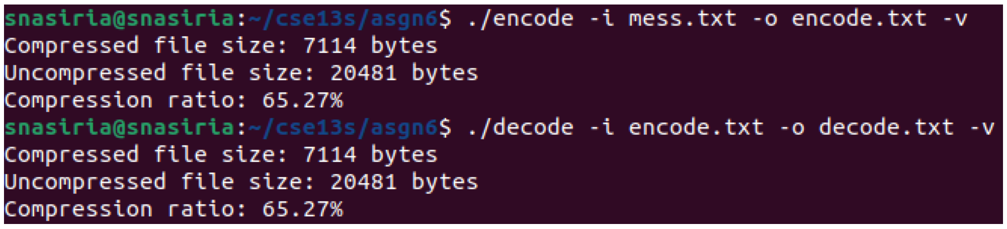
I did more testing with all kinds of different files and generally I'd be getting around 20-30% space saved with bigger files like these. With small files in the single or double digits of bytes, it is not a wise idea to use this compression because I would usually have a bigger compressed file than the original file itself. Here is an example of that:



```
snasiria@snasiria:~/cse13s/asgn6$ ./encode -i mess.txt -o encode.txt -v
Compressed file size: 65 bytes
Uncompressed file size: 54 bytes
Compression ratio: -20.37%
snasiria@snasiria:~/cse13s/asgn6$ ./decode -i encode.txt -o decode.txt -v
Compressed file size: 65 bytes
Uncompressed file size: 54 bytes
Compression ratio: -20.37%
```

Figure 3
This image demonstrates statistics of my Lempel-Ziv program with a 65B file of text. It demonstrates a negative compression ratio.

Entropy is quite important with original files because if we have a file with a lot of repetition, it greatly increases the compression ratio of the Lempel-Ziv algorithm due to its nature of using shorter codes for repeated symbols. I went ahead and generated a 20KB file with a random sequence of only the letters 'a', 'b', or 'c'. I ended up massively increasing my compression ratio to 65% due to low entropy.



```
snasiria@snasiria:~/cse13s/asgn6$ ./encode -i mess.txt -o encode.txt -v
Compressed file size: 7114 bytes
Uncompressed file size: 20481 bytes
Compression ratio: 65.27%
snasiria@snasiria:~/cse13s/asgn6$ ./decode -i encode.txt -o decode.txt -v
Compressed file size: 7114 bytes
Uncompressed file size: 20481 bytes
Compression ratio: 65.27%
```

Figure 4
This image demonstrates statistics of my Lempel-Ziv program with a 20KB file of text of only the letters "abc". It demonstrates a high positive compression ratio due to its low entropy.

This would be an example of low entropy because it has very little randomization and high predictability. In choosing a compression algorithm with high efficiency, it's important to test under the conditions of high entropy or maximized randomization like the first illustration in the previous page.

4 Citations:

I would like to cite Professor Long for the pseudocode provided in the last page of the assignment doc. I used that pseudocode to base my `encode.c` and `decode.c`.