Assignment 6 WRITEUP.pdf

Julian Chop

March 14, 2023

1 Lempel-Ziv Compression

Abraham Lempel and Jacob Ziv created two lossless compression algorithms: LZ77 and LZ78. The basic idea of these two compression algorithms is to scan the data we want to compress, look for patterns/repetitions, and essentially make a codebook out of those patterns/repetitions. We can use this codebook to compress the data as well as decompress it.

I can better explain how this idea works with an example. Say we want to compress the string "Banana". We first look at the first character "B" and compare it to any previously seen characters. Since it's the first character, we can assign "B" as the first word and assign it to the first available code, 2. The reason we want to start at to is that 1 represents an empty word. Now we check the next character in our string, "a". Because we haven't yet seen any similar characters, we assign it to code 3. We also set "n" to the next available code, 3. The next character in the string is "a" which is a character we have seen. Because we have, we set the previously seen word to "a" and look at the next character in the string, "n". We append "n" to a previously seen word and then look if we have seen this word before. We haven't. Therefore, we set "an" to the next available code, 4. Lastly, we look at the last character in the string, "a". Which is a character we have seen. Therefore, there is no need to make a new pair.

As you can see, we have essentially made a codebook for encoding the string "banana". If we wanted to decompress this encoded string, we would basically do the opposite of the steps we just took.

In this assignment, we were tasked with implementing this idea and making 2 programs that compressed data and decompressed data respectively.

2 The Trie Data Structure

I think one of the most important parts of this assignment was the use of the Trie Data Structure. Tries, also known as a prefix tree, are used in many things such as auto-complete and spelling correction. It's an efficient way to store/search a collection of words, especially when some words are just prefixes of other words.

A tree is a tree-like data structure that starts at a "root" node. Every node can have a child. And every child can have children of their own. The root represents the first character in a word. This node represents every word starting with that character. Then, that root node would have children each representing the second character in a word starting with the first character and so on.

Here's an example. Say you want to represent the word "cat" in a trie. The first letter of cat is "c" so that would be placed in the root node. Then that node would have a child node that represents "a" the second character of cat. finally, that node will also have a child representing the last character "t".

Say we now want to represent another word in the same trie, "cow". Because cow starts with the same letter as cat, we don't need to add anything to the root node. Since the second letter "o" hasn't been seen yet, we must make a new child for the root that represents the new character. Finally, that root will have a child representing "w".

In the case of this assignment, we are using a trie during compression to store words. If we can quickly search is a word is in the trie, or add them if they aren't there.

3 We Learning!

I think this assignment really brought everything we've been learning from these past assignments together. We were working with multiple ADTs which we saw in assignment 4: Game of Life. This could be seen in the TrieNode, Word, and FileHeader structs we had to manipulate. We had to work with bitwise operators to get specific bits similar to what we did in Assignment 3: Sorting: Putting your affairs in order. This could be seen in read_pair and write_pair where we had to read/write specific bits into buffers, infiles, and outfiles. Then of course pointers and dynamic allocation which we have been working with throughout the whole quarter. Other than learning about lossless compression, this assignment pushed me to use what I've learned in the previous assignments.

4 Efficiency and Entropy

Though I did not get to finish my assignment, I did do some research on the relationship between the efficiency of compression and entropy(will cite what I read in Citations). From what I understand, the entropy of a certain set of data is the amount of information it contains. The entropy of the data will determine how much you are far you are able to compress the data without losing any of it. This is called the entropy limit. Entropy isn't only determined by size, but by the randomness of the data as well. For example, it would be much easier to compress the string "abababababab", than the string "weruioohwf". Though the size is the same, the second string is more complicated, therefore having higher entropy. In conclusion, the most efficient way to compress data is to get the

closest you get to the maximum compression allowed set by the entropy of the data you're compressing, without losing any information.

5 Errors and Mishaps

This assignment was brutal, I sadly didn't get finish it completely, even with the extended time. Each of the c files we had to create wasn't that bad on their own but combining them together to implement encode and decode was ultimately the thing that got me.

One of the things I had the most trouble with was in io.c. Specifically the read_pair and write_pair. For write_pair, we had to write a pair to outfile. To do this we had to first write a code and sym to a global buffer bit by bit. Then when the buffer was full, it would write all the pairs to outfile at once. What was specifically hard about this part was keeping track of which bit in the buffer you were at, as well as writing the bits to the buffer. read_pair gave me the same issues.

Ultimately, I think the thing that gave me the most trouble was finding out where the errors were coming from. Many of the functions often relied on other functions. So an error coming from one place could be from somewhere else. I remember when I was trying to figure out why my read_pair test was returning only 1 pair when it was expecting like 172 or something like that and I was looking everywhere for the issue. It turned out that everytime I was calling read_pair, it would automatically read_bytes from the infile, even if the buffer wasn't completely read. To fix this, all I had to do was make an if statement to check if the bit_index was 0, so it would only read more bytes from infile if the buffer was flushed out.

6 Citations and Resources used

- I went to Tuesday's and Thursday's sections from 7:00-8:30 with tutor John in the first and second week of the assignment. He Helped us on how we might implement word.c as well as explain the read_pair function in io.c.
- The CSE13 discord with all the tutors was a huge help in this assignment. I could search up problems other people were having and see if others answered their questions. I get fstat() to work because of this as well as fixing up some pointer variables.
- I also used the instructions given in the .h files to code my c files. For encode and decode, I followed the pseudocode in the resources.
- to understand the trie data structure, I watched this video which visually explained how trie worked. Here's the link:

  `https://youtu.be/-urNrIAQnNo`

- to learn more about entropy and its relationship with compression efficiency, I watched/read these two things:

  `https://ieeexplore.ieee.org/document/4068914`

  `https://www.youtube.com/watch?v=M5c_RFKVkko`