Data on a Diet:
LZ77/78 Sheds Unnecessary Bits and Bytes

Lucais Sanderson

March 13, 2023

1 Introduction

The LZ77/78 compression algorithm was created by Abraham Lempel and Jacob Ziv in 1977 and 1978, respectively. Essentially, LZ saves space by consolidating repeated sequences of characters in a text/binary file and, using some type of data structure, assigns codes to each character which finally gets written as a code-symbol pair. Decoding works in reverse, retrieving the code-symbol pairs and then reconstructing the original file accordingly.

In our case, we implemented the algorithm using a Trie ADT to construct a prefix tree for encoding and a Word ADT to reconstruct the original file for decoding. Essentially, we iterate over an entire file, and if a new character with respect to the current Trie node is seen, then a new child node of the current node is created with an associated code. This code-pair is written to the file and the process continues for the entirety of the file.

For decoding, we read each code-pair created from our encoded file, save them into a Word ADT within a whole WordTable, which is later referenced to piece together longer prefixes.

2 What I Learned

2.1 The Bit Packing Problem

I'd say a primary piece I learned from this assignment was using bit manipulation to fit as much information as possible into a space. In the context of this assignment, this came in the form of representing the codes in the encoding process using the least amount of bits required. This was important because each code is initially a `uint16_t` which, by definition, requires 16 bits or 2 bytes to represent an unsigned integer. We can do better to improve the effectiveness of the compression. We are going to use up to $2^{16}-1$ codes so many of them are going to use less than the full 16 bits.

First I needed a way to find the number of bits the number required. My first thought is to use $\log_2(n)$ which would require the `math.h` library. But what made more sense was to use bit-wise operators, specifically the logical right shift. Computing this on a number until it is zero, and counting iterations yields the bits needed for a number n.

This approach makes sense because computing the right logical shift is the same as dividing the number by 2. Then we just count how many times we can do this until the number is zero which effectively tells us the biggest power of 2 the number has. Similarly, $\log_2(n)$ yields the largest power of 2 n is represented by.

Next, we need to pack a number using the number of bits specified. Because we are using an array buffer, we index it with bytes but also want to fill it in, bit by bit. A *bit index* is useful here (as specified in the spec). It would've been simple enough if we had an 8-bit number and just place it into the buffer but we have variable code sizes and aren't guaranteed byte-aligned numbers. To solve this, I worked out that using a bit-wise OR with 1 left shifted by `bit_index % 8` and the current buffer index yielded the desired result.

3 Entropy

An important variable with compression is entropy which is essentially how random a text is. For example, a set of text with low entropy would be "AAAA" because the probability we get an "A" when guessing is very high. Alternatively, "ocmqkpxjerucm" is relatively random and thus has high entropy.

We tested the LZ compression with various levels of entropy to understand its efficiency given different entropy.

3.1 High Entropy

First, we probe the algorithm's effectiveness with high entropy with sample text:

"Gxwmrjklpdbhftuvzocinaseyq Highkzxujbvwtnmpqryfsldce."

This yields -25.93% space-saving. Now this seems odd to have a negative percentage but we calculate this value by

$$100 \times (1 - \frac{\mathrm{compressed\_size}}{\mathrm{uncompressed\_size}})$$

So, we can interpret this value basically as: the higher the percentage, the more efficient.

3.2 Medium Entropy

Next, we take a look at a "medium" case:

"The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog. The quick brown fox jumps over the lazy dog."
Here, we observe a 22.22% space saving.

3.3 Low Entropy

Now we check how it reacts to a low entropy sample:

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA"

We save 81.82% with this example.

3.4 Very Low Entropy

This example is likely the simplest:

"00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000"

Our space-saving here is 78.13%.

3.5 Normal Distribution Example

Finally, we observe a more realistic case of normally distributed characters with an excerpt from "Lorem ipsum":

"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla facilisi. Fusce iaculis lacinia diam quis bibendum. Integer sit amet quam vestibulum, aliquet dolor non, dapibus nulla. Donec id nibh eu mi commodo ultrices. Fusce laoreet luctus nunc, eu imperdiet ante mollis ac. Nulla pulvinar mauris et nisi elementum lobortis. Sed bibendum turpis id urna fringilla, eget viverra lorem maximus. In euismod nibh vitae dapibus ultrices. Quisque ut placerat arcu, vel semper ante. Nulla facilisi. Nulla facilisi. Proin tristique lectus in diam molestie eleifend. Donec sagittis mi sit amet nulla aliquet dictum. Praesent pharetra lacinia turpis, ut rutrum justo convallis quis."

Our statistics say the space-saving is 21.51%.

3.6 Comparison

Accumulating our results,

Entropy	Space-Saving
High	-25.93%
Medium	22.22%
Low	81.81%
Very Low	78.13%
Random (Normal)	21.51%

It's evident that with high entropy (or randomness), the algorithm performs the worst in terms of efficiency, so much so that the compressed file is larger than the uncompressed file. This tracks since there are almost no repeated sequences to help compress the file.

As the entropy decreases, our space saving increases until we get to the very low example where the space-saving actually decreases again.

Finally, the normally distributed text yielded 21.51% space-saving which is closest to the medium sample.

4 Conclusion

In closing, I learned quite a bit about practical applications of bit-wise operations and buffers. I also developed my reasoning skills to break down each component of this algorithm.