Data compression report

Osama Housien

January 21, 2021

0.1 Report

I first used **LZSS** which is variant of the LZ77 algorithm. LZ77 is dictionary based lossless compression algorithms. Both LZ algorithms try to compress series of strings by converting the strings into a dictionary offset and string length. The general approach LZ77 follow is scan the message from first to last character, maintaining a sliding window(dictionary) of previous seen characters of length W and a look-ahead buffer of size L. The difference between the two is when encoding a string if a match is found, LZ77 will encode it as (Offset, Length, Next character) where offset represents how many characters we have to go back to reach the first character, Length represents how many characters to take from and including the first character. When no match is found LZ77 will encode it as (0,0,Character). So LZ77 algorithm encodes all strings as a length and offset, even strings with no match. So we can see that individual unmatched characters or matched strings of one or two symbols take up more space to encode than they do when uncoded . LZSS improves on LZ77 by adding a coded/uncoded flag so new token format becomes (0,charecter) for unmatched strings and (1,offset,length) for matched strings. Eg: Here we are trying to encode the string "Hell␣" using LZSS

18	.	.	.	24	25	26	27	28	.	.	.	81	82	83	84	85
W	.	.	.	H	e	l	l	o	.	.	.	H	e	l	l	␣

Sliding window (W) Look-ahead Buffer (L)

For this example I'm using L of size 5 and W size of 63. We will try to find a match for "Hell␣" in the sliding window. If no match is found we cut off the last character in the buffer. Now we are looking for the string "Hell". we see that there is a match that starts at position 20 so we set our flag to 1 and our offset will be 63 - (24-17) = 56 and the length is 4. This would give us a token (1,56,4). Now we move on and the look ahead buffer is refilled with the next 5 characters but as "␣" is the last character, Look-ahead="␣" and the sliding window now moves by 4 characters to the right.which we cant find in the sliding window so out look ahead buffer will be reduced to "" in t his case we just leave the next character uncoded so our clause will be (1,"␣") the 1 bit indicates that the next 8 bits are uncoded so we just read it as a character. In a real implementation we would want out look-ahead buffer and window size to be of size $2^N - 1$ so no bits are wasted. like 5 is 1001 in binary so we could have increased our Look-ahead to 7 decreasing the compression ratio, in fact it might increase it. Our W is of size $2^6 - 1$. "Hell␣" after encoding will be "11110001001000100000"

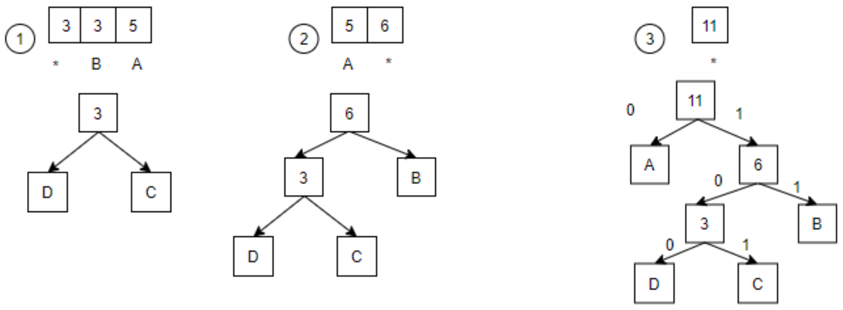
1	111000	1001	0	00100000
Flag	Offset	Length	Flag	ASCII code

In my implementation I used an offset of size $2^{15} - 1$ which means I have a sliding window (dictionary) of length 32767 and a Look-ahead buffer of size $2^6 - 1$=63. which means our total length for an encoded sequence of characters will be of size 22 bits, 1 bit for the flag and 21 for the offset and look-ahead. This means Im wasting bit or more if I encode a sequence of three characters or less therefore I only encode when I find a match of length 4 or more. Since a matching sequence has to be of length 4 or more I can make the length range artificially bigger by saying that 000000 represents 4 so when we decode the length we just add 4 to it. This gives me a max matching length 67 characters instead of 63 using same number of bits.

while LZSS reduces repetition in our file its still no where near ZIP. to improve my implementation further I decided to add **Huffman coding** which reduce statistical redundancy in the file . I will use Huffman coding to generate smaller code-words for more frequent characters in the uncoded tokens. Huffman coding works as follows: suppose we have the sequence "AABBCCAAABD". which is 88 bits to store. I first compute the frequency of each character " A:5, B:3, C:2, D:1" then insert them into a minimum priority queue.

1	2	3	5
D	C	B	A

The goal is to build a tree where each character is a leaf node. To do this we first create an empty node Z and we pop from the Queue and add the first pop to left of Z and and the the second pop to the right of Z. Then set the value of Z as the sum of both characters frequency and we insert the pair (reference to node, node value) to the queue: (* denotes reference to node)

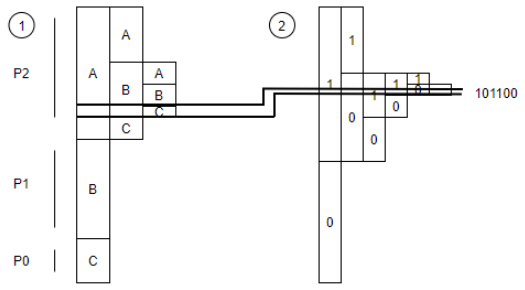


Now I can use the path from root node to a character as its unique code-words as Huffman Coding prevents any ambiguity in the decoding process using the concept of prefix code ie. a code associated with a character should not be present in the prefix of any other code. The new code-words for each letter now are "0" for A, "11" for B,"101" for C and "100" for D. Now the string "AABBCCAAABD" will take $1 \times 5 + 2 \times 3 + 3 \times 2 + 3 \times 1 = 20$ Bits instead of 88. To use Huffman in my implementation I first applied LZSS on the file then applied Huffman on the uncoded characters.

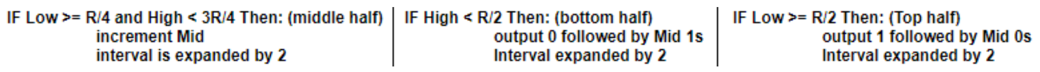
Now in order to be able to decompress the file i need to **store the Huffman tree**. To store the tree I first traverse the tree using **post order** traversal. When I encounter a leaf node I write a 1 followed by the ASCII character code. When I encounter a non-leaf node I write 0. And I append another 0 at the end to indicate end of the Huffman tree. For the example above our tree would be encoded as "1A 1D 1C 0 1B 0 0 0" 1 and 0 are bits. now in the decoder I make use of a stack to decode the tree. When 1 is read I push the next byte(represents a character) to the stack. If we read a 0 and there is one element in the stack then we are done. Otherwise we create a new node and pop top two elements of the stack and assign first one to the right child off the node and the second one to the left child. Then I push this new node to the stack. In order to use LZSS+HUFFMAN in my implementation I need to store the length of the encoded Huffman tree and since our encoded file length might not be a multiple of 8 I need to store how many padding bits I add at the end of the file so we know when to stop decoding. These 2 length along side the Huff tree will be stored at the start of the file.I use first byte to store the padded bits and the second 2 bytes to represent the tree length. After that is the encoded Huff tree and after it is the encoded text. My implementation is much better now but still preforms 10-20% worse than ZIP.

My last attempt is to use an **Preset Dictionary** alongside the LZSS dictionary. The dictionary will be of size 32768 and it will include most common English words of length 4 or more as well as most LATEX commands and signs. I will load this dictionary into a hash-table at the start. When we are looking for a match we can then look in the hash-table first then in the sliding window dictionary. This will help improve compression ratio further as for example the word "Hello" is in the hash table but not the sliding window we can still encode it. In order to use the preset dictionary I had to add one more bit to my offset so offset now is 16 bits. when a match is found in the hash-table its offset will = 2^{15}+index in the hash table. I realised we don't need to store length since offset will be a hash-table index, so I introduced a new token format (1,Offset). So when Im decoding if the offset is larger than $2^{15} - 1$ I will know that this match is from the hash table and there is no length so we jump straight to next token. This gave me an improvement anywhere between 1-5% . I believe we can improve that if we have a preset dictionary but I need to scan many files and analyse them. Overall I think this is a good implementation but its still worse than ZIP by 10-25% so I decided to leave it at that and move on to a different approach. I will include this Implementation in a folder called Old Implementation.

Lastly I decided to give PPM model with arithmetic encoding a try. **Arithmetic coding** is a near optimal coding scheme it offers a way of working at the sequence level Unlike Huffman that works at symbol level. It assigns a unique code-word for a given sequence without working out all the code-words for all sequences of the same length. For arithmetic coding to work I have to give it a list or symbols S0,S1,..,Sn and their probability occurrence P0,P1,..,Pn in the file. where $\sum$pi=1. E.g we were given symbols (A,B,C) and probabilities (0.4,0.4,0.2)



We start with interval [0,1) which means the we can use the range from 0 to 0.9$\overline{9}$, 1 is not included. Then we divide this interval among the the symbols that we have. This gives us the following intervals C: [0,0.2) B: [0.2,0.6) A: [0.6,1).We update the start interval by using these formulas **new High=Low+(High-Low)H(x),new Low=Low+(High-Low)L(x)** every time we encode a symbol.Now suppose we want to encode the sequence"ABC". first symbol is A so our new High= $0 + (1 - 0) \times 1 = 1$ and new low $= 0 + (1 - 0) \times 0.6 = 0.6$. The new main interval is[0.6,1). second symbol is B so updated interval is [0.68,0.84). Last symbol is C updated interval is [0.68,0.712). Finally the sequence "ABC" can be encoded using any value in the interval [0.68,0.712) excluding 0.712. Then we can use binary splits($\frac{1}{2^1}, \frac{1}{2^2}$,..) until we get a number that is completely contained within the interval as we see in grid 2 so "ABC" can be encoded as .101100=0.6875 so instead of using 24 bits we only used 6 bits Huffman coding would have used 7 bits. But when we want to decode we will need to to know when to stop so we can either include the of the sequence or add an EOF (end of file) symbol In my implementation I used an EOF symbol. Lets assume the C is the EOF symbol. To Decode we do the same thing we keep doing the binary splits until EOF symbol is reached. E.g 1 is read so decoded interval is [0.5,1) then 0 is read so interval is now [0.5,0.75) then 1 is read, interval is now [0.625,0.75) this interval is fully contained in the A sub interval so we output A continue like in grid 1. we keep reading bits until EOF(C) symbol is reached. But as we encode more and more symbols we start to lose precision duo to floating point errors. To avoid this we use integer representation instead. Main ideas of using integer representation are: our starting interval will be [0,R) where R=2^k(Im using K=32 In my implementation) and we use rounding to generate new interval. Then every time we update the interval



```
IF Low >= R/4 and High < 3R/4 Then: (middle half)
        increment Mid
        interval is expanded by 2

IF High < R/2 Then: (bottom half)
        output 0 followed by Mid 1s
        Interval expanded by 2

IF Low >= R/2 Then: (Top half)
        output 1 followed by Mid 0s
        Interval expanded by 2
```

To further increase my Compression ration I added a 4th order **PPM model**. In a K-th order PPM model the previous k characters are kept in a history and are used as a context, every time I read a symbol I update all the contexts of order 0,1,2...K e.g if history ="HELL" followed by "O" I increment "O" frequency in contexts "HELL", "ELL" , "LL", "L" and in order zero context. PPM makes use of conditional probabilities e.g if we have seen "what" 8 times followed by "s" 4 times. Then the conditional probability p("what"/"s")=$\frac{4}{9}$ its 9 not 8 because in my implementation every order context has an escape symbol (EOF) of frequency 1. The main idea of PPM is to reduce the context size if no match is found in the current order context. if order 0 is reached and no match has been found I just use a context of order -1 where every symbol of the Language has a frequency of 1 including the escape symbol . PPM uses these order contexts with arithmetic coding to encode the symbols. During Decoding if we decode the escape symbol using order context -1, this means end of file.

Bibliography

[1] LZSS
LZSS (LZ77) Discussion and Implementation by Michael Dipperstein
https://michaeldipperstein.github.io/lzss.html
https://blog.cloudflare.com/improving-compression-with-preset-deflate-dictionary/

[2] Huffman Coding
https://www.programiz.com/dsa/huffman-coding
https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman:~:text=To%20store%20the%20tree

[3] Arithmethic coding and PPM
https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html
https://michaeldipperstein.github.io/arithmetic.html
https://www.cs.cmu.edu/~aarti/Class/10704/Intro_Arith_coding.pdf
https://marknelson.us/posts/1991/02/01/arithmetic-coding-statistical-modeling-data-compressio
https://www.nayuki.io/page/reference-arithmetic-coding
Introduction to Data Compression, THIRD EDITION *Khalid Sayood*